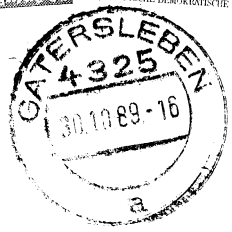
Canada



Dr. T.W. Lai
Dept. of Comput. Sci.
Waterloo Univ.
~~Ca~~ Waterloo / Ont.

Dr. H-W. Jank

Zentralinstitut für Genetik und Kulturpflanzenforschung

GDR Academy of Sciences

German Democratic Republic · Gatersleben · DDR-4325

Dear Dr. Cai

I am very much interested in your paper sent 88-26 88-32 Nov. 20/89 per Deuch

" Implicit selection "

SWAT 88, Proceedings 1988, p. 14-23

and I should greatly appreciate receiving a reprint.

With many thanks in advance for your kindness,

Sincerely yours

Dr. Hans-Wolfgang Jank

IV/18/20

To

From

Date Dec. 22/88

memo

University of Waterloo

88-26 + 88-32

Mike Anderson
EE + CS Dept.
EE + CS Bldg.
Univ of Michigan
Ann Arbor, MI.
48109-2122

sent per ~~e~~ email from
Tony Lai

Printing Requisition / Graphic Services

15047

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number,** to **extension 3451.**

TITLE OR DESCRIPTION: Space Bounds for Selection CS-88-26

DATE REQUISITIONED: July 4/88

DATE REQUIRED: ASAP

ACCOUNT NO.: 1 2 6 6 1 7 6 4 1

REQUISITIONER – PRINT: D. Wood

PHONE: 4456

SIGNING AUTHORITY: S. DeAngelis / D. Wood

MAILING INFO – NAME: Sue DeAngelis

DEPT.: C.S.

BLDG. & ROOM NO.: DC 2314

[X] DELIVER [ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 75

NUMBER OF COPIES: 100

TYPE OF PAPER STOCK: [X] BOND [ ] NCR ____ PT. [X] COVER [ ] BRISTOL [X] SUPPLIED [ ] ____

PAPER SIZE: [X] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] ____

PAPER COLOUR: [X] WHITE [ ] ____ INK: [X] BLACK [ ] ____

PRINTING: [ ] 1 SIDE ____ PGS. [X] 2 SIDES ____ PGS.

NUMBERING: FROM ____ TO ____

BINDING/FINISHING: 3 down left side

[X] COLLATING [X] STAPLING [ ] ____ PUNCHED [ ] PLASTIC RING

FOLDING/PADDING

CUTTING SIZE

Special Instructions

Math fronts and backs enclosed

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

D 0 1

D 0 1

D 0 1

TYPESETTING — QUANTITY

PAP 0 0 0 0 0 0 — T 0 1

PAP 0 0 0 0 0 0 — T 0 1

PAP 0 0 0 0 0 0 — T 0 1

PROOF

PRF

PRF

PRF

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

FLM — C 0 1

FLM — C 0 1

FLM — C 0 1

FLM — C 0 1

FLM — C 0 1

PMT

PMT — C 0 1

PMT — C 0 1

PMT — C 0 1

PLATES

PLT — P 0 1

PLT — P 0 1

PLT — P 0 1

STOCK

0 0 1

0 0 1

0 0 1

0 0 1

BINDERY

RNG — B 0 1

RNG — B 0 1

RNG — B 0 1

MIS 0 0 0 0 0 0 — B 0 1

OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL [ ] FEDERAL [ ]

GRAPHIC SERV. OCT. 85 482-2

To Sue

From Zoë

Date June 28 1988

memo

University of Waterloo

100 copies using the Math Fac TR covers;
for D Wood.

Thanks

Space Bounds for Selection

by

Tony W. Lai

Data Structuring Group
Research Report CS-88-26
June 1988

Space Bounds for Selection

by

Tony W. Lai

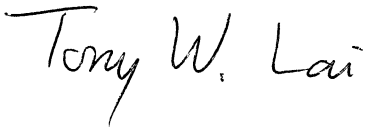
A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, 1987

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.



The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

We present several algorithms for selecting the kth smallest element from a multiset of n elements in linear time using only a constant amount of additional space. In particular, we show that $8.8256n + o(n)$ comparisons are sufficient to perform the selection if all elements are distinct, $9.3884n + o(n)$ comparisons are sufficient if elements can appear at most a constant number of times, and $10.8696n$ comparisons are sufficient in the general case. We also present an application of our implicit selection algorithms in the maintenance of implicitly represented k-d trees, and give insertion and deletions algorithms that run in $O(n \log n)$ time in the worst case.

Acknowledgements

I am very grateful to my supervisor, Prof. Derick Wood, for his direction and suggestions during the preparation of this thesis.

Contents

List of Figures

Chapter 1

Introduction

Space . . . the final frontier. — The beginning of *Star Trek*

1.1 Motivation

The *selection problem* is: determine the kth smallest element of a multiset of n elements, given the values of k and the n elements. Interest in the selection problem can be traced back to the design of tennis tournaments. In 1883, Lewis Carroll [6] published an article decrying the unfair method by which the second best player was selected; any of the players who lost to the best player may be second best. Around 1930, Hugo Steinhaus posed the question of finding the minimum number of tennis matches required to find the first and second best players. In 1932, J. Schreier [22] showed that $n + \lceil \log n \rceil - 2$ comparisons are sufficient; his method uses one knockout tournament to find the winner, and finds the second best player by holding another knockout tournament among all players who lost to the winner. S. S. Kislitsyn [13] generalized Schreier's method to any rank k by essentially using the first kth stages of a tree selection sort; Kislitsyn's method requires $n - k + \sum_{i=0}^{k-2} \lceil \log(n-i) \rceil$ comparisons. Hadian and Sobel [10] devised a method that requires only $n-k+(k-1)\lceil \log(n-k+2) \rceil$ comparisons. They hold a knockout tournament

of $n - k + 2$ players and successively eliminate the $k - 1$ players who are "too good" to be the kth best.

Blum et al. [5] devised a method that requires linear time in the worst case. Their method does not use knockout tournaments at all; we will describe their algorithm in detail in the next chapter. They devised two algorithms; the best requires at most $5.4305\dot{5}n$ comparisons in the worst case. Schönhage et al. [21] devised an algorithm for finding the median in a number of comparisons asymptotic to $3n$. Floyd and Rivest [8] devised an algorithm for selecting the kth smallest element that has an expected running time of $n + \min(n, n - k) + O(n^{1/2})$.

We define a new problem, the *implicit selection problem*, in which we want to find the kth smallest of n elements using only a constant amount of additional space. Hoare [12] devised an $O(n)$ expected time selection algorithm which can easily modified to solve the implicit selection problem in $O(n)$ expected time. We will present several algorithms that solve the implicit selection problem in $O(n)$ time in the worst case.

For our model of computation we assume a comparison-based arithmetic RAM. We will assume that comparisons have three outcomes ($<$, $=$, and $>$) and that arithmetic operations are only allowed for manipulating indices.

One motivation for the solution of the implicit selection problem comes from the update algorithms for implicitly represented k-d trees. k-d trees were first introduced by Bentley [4]. (See also Bentley [3].) Lee and Wong [14] analyzed the worst-case performance of range queries and partial range queries in k-d trees. Silva-Filho [23] analyzed the average case performance of range queries in balanced k-d trees. We will defer a formal discussion of k-d trees until Chapter 5, only giving examples of k-d trees in this chapter.

In a one-dimensional binary search tree, hereafter abbreviated as a 1-d tree, we perform searches by comparing the desired key with the key of the root, and searching either the left subtree or right subtree of the root if the keys are not equal. To construct a 1-d tree from a set of elements, we take some element p as the root, and divide the remaining elements into

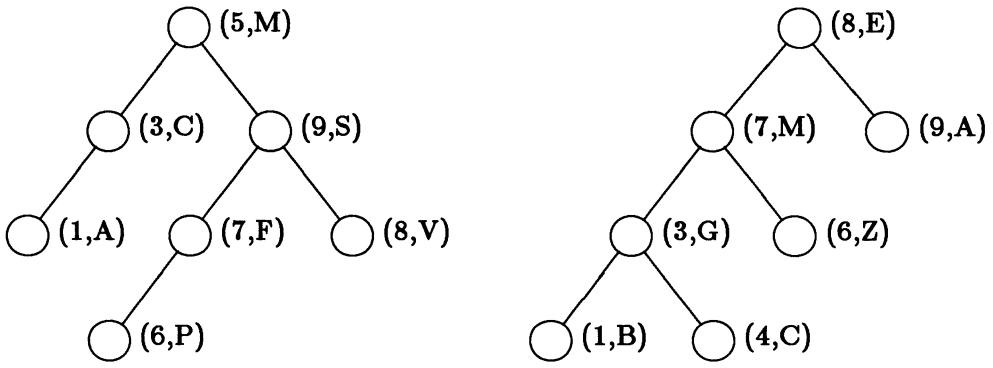


Figure 1.1: Examples of k-d trees

two subsets, based on the value of p's key. We store these two subsets as two subtrees rooted at p's children. We continue splitting subsets in this manner until each subset contains at most one element.

k-d trees are a natural k-dimensional generalization of 1-d trees. Figure 1.1 shows some examples of 2-dimensional k-d trees, hereafter referred to as 2-d trees. In a k-d tree, we cycle through the keys when performing searches, rather than using only the first key. For example, to find (6,P) in the first tree of Figure 1.1, we compare 6 to the first key of the root. Since $6 > 5$, we search the right subtree. We then compare P to the second key of node c. Since $P < S$, we move left. We then compare 6 to the first key of node e. Since $6 < 7$, we move left again. We then find that (6,P) is in node g, and we are done. See Figure 1.2.

We can construct a k-d tree from a set of elements as follows. We take some element p as the root and divide the remaining elements into two subsets based on p's first key. These two subsets are stored as two subtrees rooted as p's children. In both subsets we again take an element and divide each subset with respect to the second key. At the next level we divide with respect to the third key and so forth. After dividing with respect to the kth key, we use the first key again.

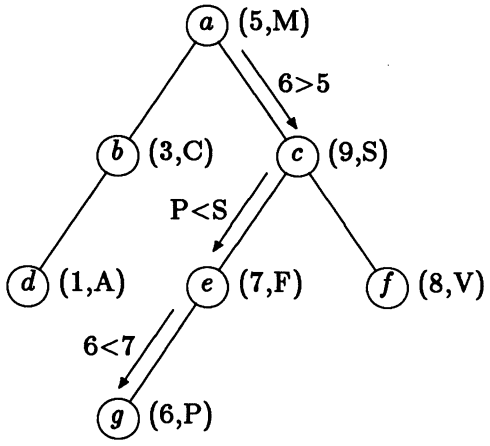


Figure 1.2: Searching for (6,P)

An *implicit data structure* is a data structure in which all elements are stored in the leftmost positions of a semi-infinite one-dimensional array, such that only a constant number of pointers need be kept. No distinction is made between a pointer and an integer index in the range $[0, n]$, where n is the number of elements. Implicit data structures were first explicitly studied by Munro and Suwanda [19]. Two simple examples of implicit data structures are sorted lists and heaps.

Various implicit data structures have been devised since implicit data structures were first studied, but most are only one-dimensional. Munro [17] devised a static implicit data structure based on k-d trees. Van Leeuwen and Wood [25] devised an implicit d-dimensional dynamic data structure that can solve complementary range queries in constant time plus the time required to report the answers. Munro [18] devised another static implicit data structure that can solve partial match queries in polylogarithmic time.

A *left-complete binary tree* is a binary tree in which all leaves appear on two adjacent levels and are as far to the left as possible. We can easily represent a left-complete k-d tree implicitly by storing the nodes of the tree in breadth-first order. Figure 1.3 gives an example of this mapping.

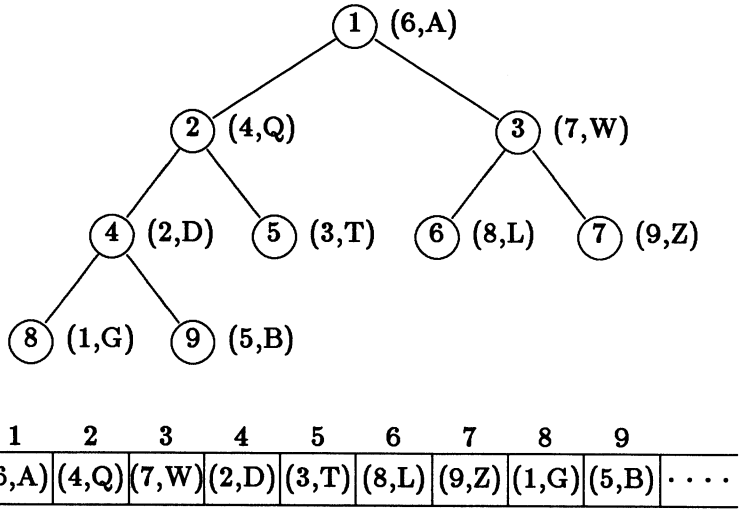


Figure 1.3: Representing a k-d tree implicitly

This representation is basically static. If no restrictions are placed on the balance of the k-d tree, then insertions can be performed in a straightforward manner. We simply perform an exact match query on the k-d tree and insert the node where the query fails; this is analogous to insertions in a 1-d tree. However, this method is clearly inadequate to maintain left-completeness for our implicit representation.

To perform updates, we *dynamize* our data structure by exploiting our construction algorithm for implicit left-complete k-d trees. Bentley [1] was the first to propose a scheme for transforming a static data structure into a semidynamic data structure. Van Leeuwen and Wood [24] introduced the term "dynamization" and gave a method for transforming static structures to fully dynamic structures. Bentley [2] and Overmars [20] have suggested various other dynamization schemes.

One method of maintaining left-completeness is to insert the node using the above method and then reconstruct the smallest subtree necessary to guarantee left-completeness. Suppose we want to insert the element (0,C) into the 2-d tree depicted in Figure 1.3. See Figure 1.4. Consider the root x of the subtree we have to rebuild. We clearly have to rebuild

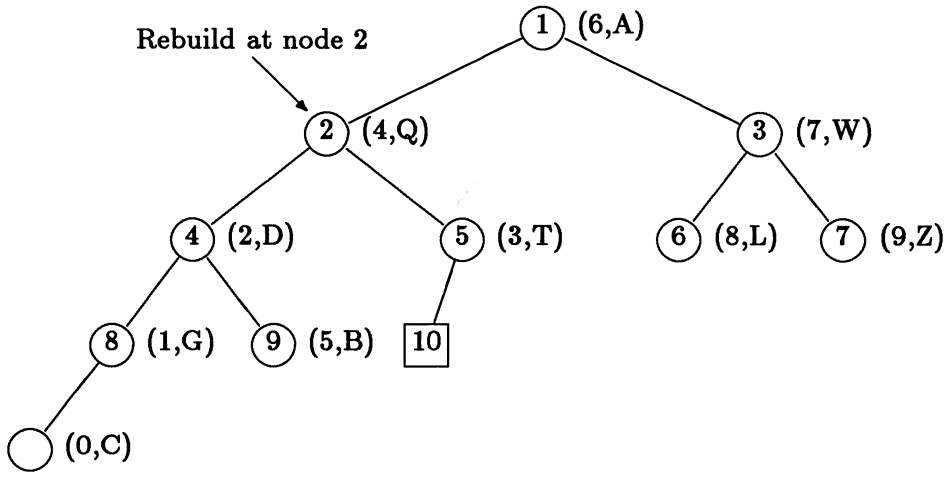


Figure 1.4: Inserting into a left-complete k-d tree

a subtree rooted at an ancestor of the inserted node. The subtree must also be rooted at an ancestor of the next free space, indicated by the square leaf, since this position must be occupied in a left-complete tree of 10 nodes. Thus, a candidate for x is the lowest common ancestor of the inserted node and the next free space.

To reconstruct this subtree, we want to select the fourth smallest element of the subtree with respect to the second key, so that three elements will fall in the left subtree and two elements will fall in the right subtree. In general, for any node x we will select the $(n_L + 1)$th element, where n_L is the number of nodes in the left subtree of x, with respect to the appropriate key. We can always determine n_L since the shape of a left-complete subtree is solely dependent on the number of elements n in the data structure. Allowing the insertion algorithm to use more than a constant amount of additional space defeats the purpose of using an implicit data structure, so we do not want the selection algorithm to use more than a constant amount of extra storage. This provides one motivation for the implicit selection problem. The construction of interval *heaps provides another motivation; see [25].

In chapter 5, we will discuss the insertion and deletion algorithms for our data structure in detail.

1.2 Summary

In chapter 2, we discuss two algorithms for solving the implicit selection problem for distinct elements. We describe how constant additional space can be achieved, and give a simple algorithm that requires $27.38462n + o(n)$ comparisons. We then optimize this to obtain an algorithm that requires $14.8n + o(n)$ comparisons. We also show that the number of data movements required by our algorithms is $O(n)$.

In chapter 3, we discuss solution of the implicit selection problem for arbitrary elements. We describe new problems introduced by the possibility of repetitions and the solutions to these problems. We then give an algorithm that requires $\frac{501}{13}n \approx 38.5385n$ comparisons and optimize it to obtain an algorithm that requires $\frac{383.6}{13}n \approx 29.5077n$ comparisons. We again show that the number of data movements required by our algorithms is $O(n)$.

In chapter 4, we discuss some improvements to our algorithms that allow us to reduce the number of comparisons to $8.8256n+o(n)$ for distinct elements and $10.8696n$ for arbitrary elements. The number of data movements required by these algorithms is again shown to be $O(n)$.

In chapter 5, we give an application of implicit selection in the maintenance of implicit k-d trees. We give an algorithm to implicitly solve any intersection query, and we analyze the worst-case time, the amortized worst-case time, and the expected time of our insertion and deletion algorithms.

Finally, in chapter 6, we give some conclusions and discuss some open problems.

Chapter 2

Implicit Selection for Distinct Elements

2.1 The Blum-Floyd-Pratt-Rivest-Tarjan algorithm

In the selection problem, we want to select the kth smallest of n elements. The implicit selection problem is similar except that we are restricted to using only constant additional space. There are two variants of this problem: we can require that the n elements be distinct, or we can place no restriction on the elements.

In this chapter, we will discuss an algorithm that performs implicit selections in linear time but requires that all elements be distinct. The next chapter discusses an implicit selection algorithm that places no such restriction.

Blum et al. devised two selection algorithms that require $\Theta(n)$ time in the worst case. They devised a simple, "slow" algorithm that requires $19.\dot{3}n$ comparisons and a complicated, "fast" algorithm that requires $5.4305\dot{5}n$ comparisons.

Our new selection algorithms are based on the slow Blum-Floyd-Pratt-Rivest-Tarjan algorithm, hereafter referred to as the BFPRT algorithm, and so we will discuss its op-

eration. Let c be some constant, where $c \geq 5$. Let $\#S$ denote the size of a multiset S. Then the BFPRT algorithm computes the kth smallest element of a multiset S is as follows:

function BFPRT-SELECT(S,k)

1. Arrange S into $\lfloor n/c \rfloor$ lists of c elements, and sort each list.
2. Let T be the set of medians from each of the lists. Compute $m =$ BFPRT-SELECT($T, \lceil \frac{n}{2c} \rceil$).
3. Let $S_<$, $S_=$, and $S_>$ be the sets of elements less than, equal to, and greater than m, respectively.
4. If $k \leq \#S_<$, then return BFPRT-SELECT($S_<$,k). If $\#S_< < k \leq \#S_< + \#S_=$, then return m. Otherwise, return BFPRT-SELECT($S_>$,$k - \#S_< - \#S_=$).

end BFPRT-SELECT

Clearly this algorithm requires $O(n)$ space since it requires $O(n)$ time. Hence it is space-optimal using the usual definition of space cost. However, Blum et al. did not consider the implicit selection problem and did not pursue a method for implementing the algorithm using $O(1)$ additional storage.

2.2 Achieving constant space

There are four factors that contribute to the storage requirements of the BFPRT algorithm, the last three being due to the recursion:

1. Recopying space
2. Saving arguments—endpoints
3. Saving arguments—k

4. Implementing recursive calls

We will show in turn how each of these costs can be reduced to a constant.

Saving function values may also appear to contribute to the storage requirements of the BFPRT algorithm, but function values never have to be saved. This is because in the BFPRT algorithm, the result of the first recursive call in step 2 is discarded before the second recursive call in step 4, which implies that function values never exist simultaneously.

2.2.1 Recopying space

Although step 2 of the BFPRT algorithm may appear to require the recopying of $O(n)$ elements, it can be implemented without using any recopying, so that it requires $O(\log n)$ additional space for the recursion. One method of avoiding recopying is to ensure that selections are only performed on contiguous sequences of elements.

This can be achieved as follows. In step 1 of the algorithm, we scatter the elements of the lists such that list i, for $1 \leq i \leq \frac{n}{c}$, consists of the elements in the locations $i, \frac{n}{c} + i, \frac{2n}{c} + i, \ldots, (c-1)\frac{n}{c} + i$. This guarantees that after sorting each list, the medians fall into a contiguous block occupying locations $\lfloor \frac{c-1}{2} \rfloor \frac{n}{c} + 1$ through $\lceil \frac{c}{2} \rceil \frac{n}{c}$. To partition S into $S_<$, $S_=$, and $S_>$ as required in step 4, we use a modified quicksort partition algorithm [11] to ensure that each set occupies a contiguous block.

Therefore, no elements have to be recopied into a separate array, and only the recursion contributes to the additional space cost. We will restate the BFPRT algorithm in terms of an input array A, lower and upper bounds l and u, and a rank k.

function RSELECT(A,l,u,k)

1. Arrange $A[l \ldots u]$ into $\frac{u-l+1}{c}$ scattered lists of size c, and sort each list.
2. Find $m = \text{RSELECT}(A, l + \lfloor \frac{c-1}{2} \rfloor \frac{n}{c}, l + \lceil \frac{c}{2} \rceil \frac{n}{c} - 1, \lceil \frac{u-l+1}{2c} \rceil)$.

3. Partition $A[l \ldots u]$ so that there are positions l', u' such that every element in $A[l \ldots l'-1]$ is less than x, every element in $A[l' \ldots u']$ is equal to x, and every element in $A[u'+1 \ldots u]$ is greater than x.

4. If $k \leq l' - l$, then return RSELECT$(A, l, l'-1, k)$. If $l' - l < k \leq u' - l + 1$, then return m. Otherwise, return RSELECT$(A, u'+1, u, k-(u'+1-l))$.

end RSELECT

Note that in step 1 we only need to find the median of each list of size c; sorting is of no additional benefit. Thus we can replace the sorting in step 1 by the Hadian-Sobel median finding algorithm.

2.2.2 Saving arguments—endpoints

RSELECT requires that we be able to retrieve the endpoints of previously processed lists, and there can be $O(\log n)$ such lists. We cannot recover the endpoints of previous lists by knowledge of the endpoints of the currently processed list alone, so we need to store the endpoints of previous lists in some way without using too much additional space. Because we are only permitted to use constant additional space, our only option is to implicitly encode the endpoints of the current list in the input array.

There is a surprisingly simple method for preserving the current range bounds. Our method is inspired by the encoding method used by Durian's stackless quicksort [7], but our scheme bears no resemblance to Durian's scheme. The basic idea of the method is as follows. Let α be some list in an array A, that is, let α be some set of contiguous elements of A, and let $left_\alpha$ and $right_\alpha$ denote the left and right bounds of α. Let β be a proper sublist of α, and let γ be a proper sublist of β, that is, $left_\alpha < left_\beta < left_\gamma$ and $right_\gamma < right_\beta < right_\alpha$. See Figure 2.1. Let α_{min} and α_{max} be the minimum and maximum elements of α, respectively. If $\alpha_{min} \notin \beta$ and $\alpha_{max} \notin \beta$, then the assumption of element distinctness guarantees that $\alpha_{min} < \beta_{min}$ and $\alpha_{max} > \beta_{max}$. Suppose $A[left_\beta - 1] = \alpha_{min}$, $A[right_\beta + 1] = \alpha_{max}$,

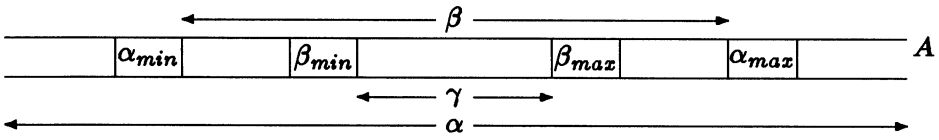


Figure 2.1: The encoding method

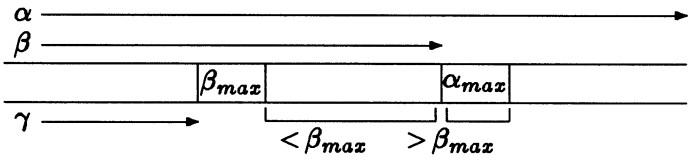


Figure 2.2: The invariants of the encoding method

$A[left_\gamma - 1] = \beta_{min}$, and $A[right_\gamma + 1] = \beta_{max}$. Notice that any element positioned between β_{max} and α_{max} must be a member of β and is thus less than β_{max}. Therefore a search for the first element to the right of $\beta_{max} = A[right_\gamma + 1]$ that is greater than β_{max} will suffice to find $\alpha_{max} = A[right_\beta + 1]$, and thus from $right_\gamma$ we can find $right_\beta$. Refer to Figure 2.2. $left_\beta$ can be found from $left_\gamma$ in a similar manner. If we are currently processing a list α, and we want to process a sublist β of α, then we store the endpoints of α by placing α_{min} in the left end of β (i.e. $A[left_\beta - 1]$) and α_{max} in the right end of β (i.e. $A[right_\beta + 1]$).

It should be clear that we can extend the above scheme indefinitely. Figure 2.3 shows how the endpoint markers are positioned with several nested lists. Figure 2.4 shows the inequalities that hold with several nested lists.

We can formalize the above scheme and prove its correctness as follows. We will use L_i and R_i to denote the left and right bounds of list i, and l to denote the number of lists.

Theorem 2.1 *Suppose we have an array A and indices* $L_1, L_2, \ldots, L_l, R_1, R_2, \ldots, R_l$ *such that:*

(i) $L_1 < L_2 < \cdots < L_l < R_l < R_{l-1} < \cdots < R_1$ *(The lists are nested.)*

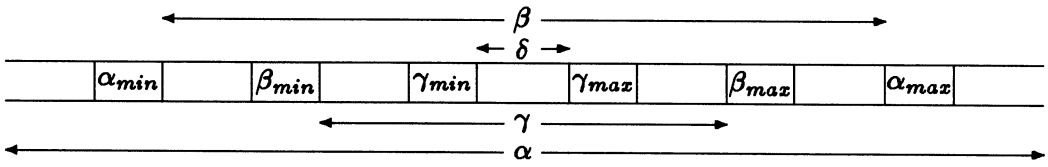


Figure 2.3: Encoding nested lists

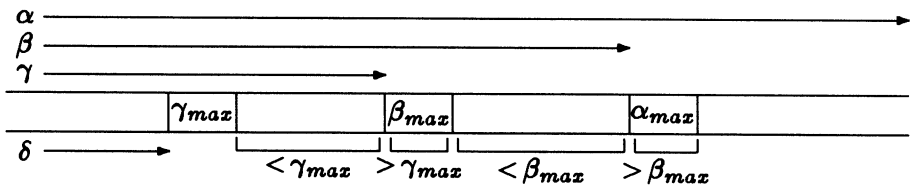


Figure 2.4: The invariants for encoding nested lists

(ii) $\forall i, \forall j, (A[i] = A[j] \Longrightarrow i = j)$ *(All elements are distinct.)*

(iii) $\forall i, \forall j \ni L_i \leq j \leq R_i,\ A[L_i - 1] < A[j] < A[R_i + 1]$ *(Each list is surrounded by two elements that are strictly less than and greater than all of the elements in the list.)*

(iv) $\forall i < l, (A[L_{i+1} - 1] = \min_{L_i \leq k \leq R_i}\{A[k]\}) \wedge (A[R_{i+1} + 1] = \max_{L_i \leq k \leq R_i}\{A[k]\})$ *(The elements surrounding list $i + 1$ are the minimum and maximum elements of list i.)*

Then

(I) $\forall i < l, \forall j \ni L_i - 1 \leq j < L_{i+1} - 1, (A[j] < A[L_{i+1} - 1] \Longleftrightarrow j = L_i - 1)$

(II) $\forall i < l, \forall j \ni R_{i+1} + 1 < j \leq R_i + 1, (A[j] > A[R_{i+1} + 1] \Longleftrightarrow j = R_i + 1)$

(III) *If* $\exists L_{l+1}, R_{l+1} \ni L_l < L_{l+1} < R_{l+1} < R_l$, *then we can rearrange some elements in the set* $\{A[L_l], \ldots, A[R_l]\}$ *such that conditions (i) through (iv) are satisfied for indices* $L_1, \ldots, L_{l+1}, R_1, \ldots, R_{l+1}$.

Proof:

(I) Let i and j be arbitrary integers such that $i < l$ and $L_i - 1 \leq j < L_{i+1} - 1$. It is sufficient to show

1. $j = L_i - 1 \Longrightarrow A[j] < A[L_{i+1} - 1]$

   Condition (i) implies $L_i < L_{i+1} < R_i$, which implies that $L_i \leq L_{i+1} - 1 < R_i$. But condition (iii) implies that $A[j] = A[L_i - 1] < A[L_{i+1} - 1]$.

2. $j > L_i - 1 \Longrightarrow A[j] \geq A[L_{i+1} - 1]$

   Condition (iv) implies that $A[L_{i+1} - 1] = \min_{L_i \leq k \leq R_i}\{A[k]\}$. But from the definition of min we know that $\forall j \ni L_i \leq j \leq R_i$, $A[j] \geq A[L_{i+1} - 1]$. Condition (i) implies that $\forall j \ni L_i \leq j < L_{i+1}$, $A[j] \geq A[L_{i+1} - 1]$; this implies that $\forall j \ni L_i - 1 < j \leq L_{i+1} - 1$, $A[j] \geq A[L_{i+1} - 1]$.

Since i and j are arbitrary, $\forall i < l, \forall j \ni L_i \leq j \leq R_i$, $A[j] < A[L_{i+1} - 1] \Longleftrightarrow j = L_i - 1$. Note that condition (ii), the assumption of element distinctness, is not necessary to prove (I) and (II).

(II) The proof of (II) is analogous to the proof of (I).

(III) Let L' and R' be the indices of the minimum and maximum elements of $\{A[L_l], \ldots, A[R_l]\}$, respectively. We will show that conditions (i) through (iv) are satisfied if we swap $A[L_{l+1} - 1] \leftrightarrow A[L']$ and $A[R_{l+1} + 1] \leftrightarrow A[R']$.

Since $L_1, \ldots, L_l, R_1, \ldots, R_l$ are not affected and $L_l < L_{l+1} < R_{l+1} < R_l$ by hypothesis, condition (i) must hold. Also, swapping elements cannot affect element distinctness, so condition (ii) is satisfied.

Note that condition (iii) held for list bounds $L_1, \ldots, L_l, R_1, \ldots, R_l$ and that swapping elements in $\{A[L_l], \ldots, A[R_l]\}$ does not affect the validity of condition (iii) for the above list bounds. Thus to show condition (iii) holds for $L_1, \ldots, L_{l+1}, R_1, \ldots, R_{l+1}$, it is sufficient to show that $\forall j \ni L_{l+1} \leq j \leq R_{l+1}, A[L_{l+1} - 1] < A[j] < A[R_{l+1} +$

1]. But $A[L_{l+1} - 1]$ and $A[R_{l+1} + 1]$ are the minimum and maximum elements of $\{A[L_l], \ldots, A[R_l]\}$, respectively, so $\forall j \ni L_{l+1} \leq j \leq R_{l+1}$, $A[L_{l+1} - 1] \leq A[j] \leq A[R_{l+1} + 1]$. But condition (ii) guarantees that no elements in different positions are equal, implying that $\forall j \ni L_{l+1} \leq j \leq R_{l+1}, A[L_{l+1} - 1] < A[j] < A[R_{l+1} + 1]$. Thus condition (iii) holds.

Condition (iv) also held for list bounds $L_1, \ldots, L_l, R_1, \ldots, R_l$ and swapping elements in $\{A[L_l], \ldots, A[R_l]\}$ also does not affect the validity of condition (iv). Thus to show that condition (iv) holds for $L_1, \ldots, L_{l+1}, R_1, \ldots, R_{l+1}$, it is sufficient to show that $A[L_{l+1} - 1] = \min_{L_l \leq k \leq R_l}\{A[k]\}$ and $A[R_{l+1} - 1] = \max_{L_l \leq k \leq R_l}\{A[k]\}$. But the exchanges that we performed guarantee this. Thus condition (iv) holds.

□

The significance of (I) is that we can always find L_i from L_{i+1} for all $i < l$ by searching for the first element to the left of L_{i+1} that is smaller than L_{i+1}. Similarly, (II) implies that we can always find R_i from R_{i+1} for all $i < l$. The significance of (III) is that our proposed scheme can be continued more or less indefinitely. This proves the validity of our proposed scheme for encoding list bounds.

2.2.3 Saving arguments—k

Another problem is that we have to implicitly encode the rank k that we are selecting. This need only be done for the first recursive call of the BFPRT algorithm, that is, in step 2, since the second recursive call in step 4 is the last action taken by the BFPRT algorithm.

Encoding k is straightforward if all elements are distinct; we can use a binary encoding scheme. If a list α is of size n, then since $1 \leq k \leq n$, we can encode k in the relative order of the first $2\lceil \log n \rceil$ elements of α. We use a pair of elements to encode each bit of k: we place the pair in ascending order to indicate a 0 and in descending order to indicate a 1.

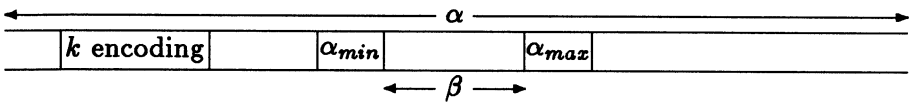


Figure 2.5: Encoding k

This always allows us to recover the stored bit since no two elements are equal. Note that we always use $2\lceil \log n \rceil$ elements to encode k; leading zero bits are kept.

k has to be encoded only during step 2 of the BFPRT algorithm, and only the middle n/c elements can be reordered in step 2. Thus, if $2\lceil \log n \rceil < \lfloor \frac{c-1}{2} \rfloor \frac{n}{c} - 1$, then the elements used to encode k cannot be affected. Clearly the above inequality is satisfied for all $n \geq 64$, and we can sort the elements if $n < 64$. Also note that once we recover the endpoints of a list α, the size of α is sufficient to determine the elements used to encode k, and so we can always recover k.

Figure 2.5 shows how the elements of a list are rearranged to encode l, u, and k.

2.2.4 Implementing recursive calls

We can use a stackless iterative routine that conceptually performs a postorder traversal of the recursion tree of *RSELECT*. Moving left or right in the recursion tree corresponds to solving the recursive calls of *RSELECT*. Moving up the recursion tree is possible because of our encoding and recovery schemes.

However, once we solve a selection subproblem, we need some means of distinguishing whether the subproblem corresponds to step 2 or step 4 of *RSELECT* because we need to determine whether we are going up from a left or right child of the recursion tree. We can distinguish between the subproblems by comparing the endpoints of the lists that we have returned from and are currently processing. Suppose that we have returned from processing a list β to processing a list α. If we have just solved the first subproblem, then β is the

middle n/c elements of α. If we have solved the second subproblem, then we require that either $left_\beta = left_\alpha + 1$ or $right_\beta = right_\alpha - 1$.

2.3 The new algorithm

2.3.1 The algorithm

There are still one complication that we have not discussed. Because of the manner in which our encoding scheme works, we have to deal with the special cases of selecting the minimum and the maximum separately. This is because α_{min} and α_{max} are used to encode the bounds of the list α when processing a sublist β of α. Handling the cases of selecting the minimum and maximum is straightforward.

Our algorithm can be described as follows. Let n_0 be some sufficiently large constant.

function DistinctISELECT(A,l,u,k)

1. (Initialization.)
   Set $l_0 \leftarrow l$ and $u_0 \leftarrow u$.
2. (Handle special cases.)
   If $k = 1$, then set $x \leftarrow \min(A, l, u)$ and go to 6.
   If $k = u - l + 1$, then set $x \leftarrow \max(A, l, u)$ and go to 6.
   If $u - l < n_0$, then sort $A[l \ldots u]$, set $x \leftarrow A[l + k - 1]$, and go to 6.
3. (Solve the first subproblem.)
   Find minimum and maximum of $A[l \ldots u]$, and swap them with $A[l]$ and $A[u]$, respectively.
4. Arrange $A[l + 1 \ldots u - 1]$ into $\lfloor \frac{u-l-1}{c} \rfloor$ scattered lists of size c, and place the median of each list in the middle of the list.

5. Let l', u' be the endpoints of the list of medians. Encode l and u by swapping $A[l] \leftrightarrow A[l'-1]$, $A[u] \leftrightarrow A[u'+1]$. Encode k in the relative order of $A[l+1 \ldots l+2\lfloor \log(u-l) \rfloor + 2]$. Set $l \leftarrow l'$, $u \leftarrow u'$, $k \leftarrow \lceil \frac{u'-l'+1}{2} \rceil$. Go to 2.

6. (The termination condition.)
   If $l = l_0$ and $u = u_0$, then return x.

7. (The recovery step.)
   Set $l' \leftarrow l$, $u' \leftarrow u$. Recover l and u as follows. Find the index i of the first element to the left of $A[l'-1]$ that is less than $A[l'-1]$. If no such i exists, then set $l \leftarrow l_0$; otherwise, set $l \leftarrow i+1$. Find the index j of the first element to the right of $A[u'+1]$ that is greater than $A[u'+1]$. If no such j exists, then set $u \leftarrow u_0$; otherwise, set $u \leftarrow j-1$.

8. (Returned from second subproblem, so the current subproblem is finished.)
   If $l = l'-1$ or $u = u'+1$, then go to 6.

9. (Returned from first subproblem—solve the second subproblem.)
   Recover k. Swap $A[l'-1] \leftrightarrow A[l]$, $A[u'+1] \leftrightarrow A[u]$.

10. Partition $A[l+1 \ldots u-1]$ so that there is a position p such that $A[p] = x$, everything in $A[l+1 \ldots p-1]$ is less than x, and everything in $A[p+1 \ldots u-1]$ is greater than x.

11. If $k = p-l+1$, then go to 6.
    If $k \leq p-l$, then set $l' \leftarrow l+1$, $u' \leftarrow p-1$, $k \leftarrow k-1$.
    If $k > p-l+1$, then set $l' \leftarrow p+1$, $u' \leftarrow u-1$, $k \leftarrow k-1-(p-l)$.
    Encode l and u by swapping $A[l] \leftrightarrow A[l'-1]$, $A[u] \leftrightarrow A[u'+1]$. Set $l \leftarrow l'$, $u \leftarrow u'$.
    Go to 2.

end DistinctISELECT

2.3.2 Correctness

To show the correctness of *DistinctISELECT*, we will show that it emulates a slightly modified version of *RSELECT*. More specifically, we consider a modified version of *RSELECT* where we precede the first step by a step 0 that handles special cases as in step 2 of *DistinctISELECT*. We will refer to this modified version of *RSELECT* as *RSELECT2*.

We claim that *DistinctISELECT* performs an postorder traversal of the recursion tree of *RSELECT2*; that rearrangement in *DistinctISELECT* is identical to that of *RSELECT2*; that l, u, and k are properly set on entry to subproblems; that l, u, and k are correctly recovered; and that the output x is propagated correctly. Step 0 of *RSELECT2* is identical to step 1 of *DistinctISELECT*, so step 0 is obviously correctly emulated. Step 1 of *RSELECT2* only rearranges the elements. The emulation of step 2 of *RSELECT2* depends only on the correctness of the assignment and recovery of l, u, and k and the correct propagation of x. Step 3 also only rearranges elements. Step 4 depends on the correctness of the assignment of l, u, and k; the recovery of l and u; and the correct propagation of x. Thus the validity of our claims implies the correctness of the emulation.

First, notice in the recursion tree, if a node has any children then it has a left child. Hence to traverse the recursion tree using the operations move left, move right, and move up, we can do the following.

(I) Move left until we reach a leaf.

(II) If we are at the root, then quit. Otherwise, move up. If we have moved up from a right child, then go to (II). If we have moved up from a left child, then move right and go to (I).

Steps 2–5 of *DistinctISELECT* move left in the recursion tree until a leaf is reached. Steps 6 and 7 move upward, quitting if we are at the root. Step 8 goes to step 6 if we move up from a right child, and steps 9–11 move right and go to step 2 if we move up from a left child. Thus steps 2–5 correspond to (I) and steps 6–11 correspond to (II). Note that

from the previous section we know that our method of determining the child which we have moved up from is correct.

Clearly steps 4 and 10 of our algorithm are identical to steps 1 and 3 of *RSELECT2*, respectively; thus the rearrangement performed by *DistinctISELECT* is correct.

Clearly steps 5 and 11 correctly set l, u, and k on entry to the first and second subproblems, respectively. Also note that our encoding of l and u satisfy Theorem 2.1 and our encoding of k is valid from Section 2.2.3.

By Theorem 2.1, the linear searches performed in step 7 are sufficient to find l and u. Also, k is recovered only during the first subproblem, and our method of recovering k is valid from Section 2.2.3. Thus l, u, and k are recovered correctly.

Finally, step 2 of *DistinctISELECT* finds a value of x and goes up the recursion tree immediately if we are selecting the maximum or minimum or if $u - l$ is sufficiently small. This corresponds to step 0 of *RSELECT2*. After returning from a first subproblem, we use x in step 10 and discard it before solving the second subproblem, which is identical to the use of function outputs in steps 3 and 4 of *RSELECT2*. When returning from a second subproblem, we propagate x upward, corresponding to how step 4 of *RSELECT2* returns the function value of its second recursive call.

2.3.3 Analysis

We derive a recurrence relation to measure the cost of *DistinctISELECT* independently of the cost measure. Consider how the recursion tree is traversed in our algorithm. Each node of the tree corresponds to a subproblem of the algorithm. We can associate the cost of preparing and recovering from an instance x with the edge $(x, parent(x))$, and we can associate the cost of solving a leaf instance x with the leaf itself. We let the cost of an internal node y be the sum of the costs of all edges and leaves in the subtree rooted at y, so that the cost of the root of the recursion tree is the total cost of the algorithm. Note that the cost of an internal node is just the sum of the costs of its children and the edges between

it and its children. Hence we can write the cost of *DistinctISELECT* as the recurrence relation

$$T(n) = T(s_1(n)) + T(s_2(n)) + f_{1,p}(n) + f_{1,r}(n) + f_{2,p}(n) + f_{2,r}(n)$$

where $s_1(n)$ is the size of the list of subproblem 1, $s_2(n)$ is the size of the list of subproblem 2, $f_{i,p}(n)$ is the cost of preparing for subproblem i, and $f_{i,r}(n)$ is the cost of recovering from subproblem i. If we are measuring the total cost of all operations of *DistinctISELECT*, then $s_1(n) = \frac{n}{c}$, $s_2(n) \leq (1 - \lceil \frac{c}{2} \rceil / 2c)n$, and for all i, $f_{i,p}(n)$ is $O(n)$ and $f_{1,r}(n)$ is $O(n)$. Clearly for $c \geq 5$, there exists a constant $d < 1$ such that $s_1(n) + s_2(n) < dn$, which implies $T(n)$ is $O(n)$.

We now count the number of comparisons more carefully, using the above recurrence relation for $T(n)$. We let $c = 29$; this is the optimal value of c. Then $s_1(n) = \frac{n}{29}$ and $s_2(n) \leq \frac{43}{58}n$. To set up the first subproblem, we need to find the maximum and minimum, requiring $\frac{3}{2}n$ comparisons; find medians of $\frac{n-2}{29}$ lists of size 29, requiring about $\frac{70}{29}n$ comparisons using Hadian and Sobel's selection algorithm [10]; encode l and u, requiring no comparisons; and encode k, requiring $2\lceil \log n \rceil$ comparisons. Thus $f_{1,p}(n) = \frac{3}{2}n + \frac{70}{29}n + 2\lceil \log n \rceil = \frac{227}{58}n + 2\lceil \log n \rceil$. To recover l, u, and k, we need $f_{1,r}(n) = n - s_1(n) + 2\lceil \log n \rceil = \frac{28}{29}n + 2\lceil \log n \rceil$ comparisons. To set up the second subproblem, we need to partition the elements, requiring n comparisons, and encode l and u, requiring no comparisons, so that $f_{2,p}(n) = n$. To recover from the second subproblem, we need to find l and u, requiring $n - s_2(n)$ comparisons. Thus the number of comparisons required is

$$\begin{aligned} C(n) &= C(\frac{n}{29}) + C(s_2(n)) + f_{1,p}(n) + f_{1,r}(n) + f_{2,p}(n) + f_{2,r}(n) \\ &= C(\frac{n}{29}) + C(s_2(n)) + \frac{399}{58}n + 4\lceil \log n \rceil - s_2(n) \end{aligned}$$

Since $C(n) > n$, $C(s_2(n)) - s_2(n)$ is maximized by setting $s_2(n)$ as large as possible. Thus we have the recurrence relation

$$C(n) = C(\frac{n}{29}) + C(\frac{43}{58}n) + \frac{356}{58}n + 4\lceil \log n \rceil$$

We can show by induction that $C(n) \leq \frac{356}{13}n + o(n) < 27.38463n + o(n)$. The basis for the induction is that we can select from $n < 64$ elements by sorting; our encoding schemes for l, u, and k are guaranteed to work when $n \geq 64$. Note that the choice of the induction basis corresponds to the choice of the constant n_0 used by *DistinctISELECT*. To minimize the $o(n)$ term, n_0 should be set to 10^8.

We also count the number of data movements performed by *DistinctISELECT*. To set up the first subproblem, we need 6 data movements to move the minimum and maximum elements, $\frac{3}{29}n$ movements to move the medians, 6 movements to encode l and u, and $6\lceil \log n \rceil$ movements to encode k. Recovering from the first subproblem requires no data movements. To set up the second subproblem, we need 6 movements to move the minimum and maximum, $\frac{3}{2}n$ movements to partition the elements, and 6 movements to again move the minimum and maximum. Recovering from the second subproblem requires no data movements. Thus the number of data movements required is

$$M(n) = M(\frac{n}{29}) + M(\frac{43n}{58}) + \frac{3n}{2} + \frac{3n}{29} + 6\lceil \log n \rceil + 24$$

By induction, we can show that $M(n) \leq \frac{93}{13}n + o(n) < 7.15385n + o(n)$. The basis for the induction is that we can sort the elements when $n \leq 64$ using an auxiliary array of pointers.

2.4 Optimizations

We can make a one-endpoint version of our algorithm in that we can fix the left bound of any processed list to be the left end of the input array, so that only the right bound can vary. To see how this is possible, first notice that we can make the set of medians in step 2 occupy a contiguous block starting at location 1 by moving the median of each scattered list to the first location of the list. Second, we consider steps 4 and 5 to be discarding elements rather than partitioning elements, and we always keep retained elements at the left end of the current list. Then only one endpoint has to be encoded. We can use an asymmetric

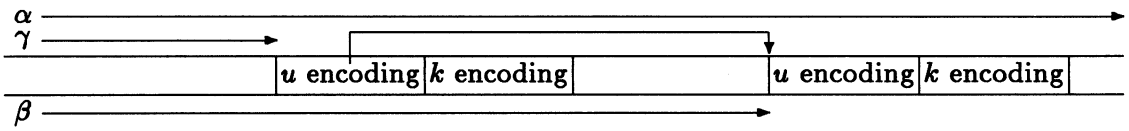


Figure 2.6: Using one endpoint

version of the bound-encoding scheme presented earlier, but we can make a much more efficient algorithm by encoding u using the relative order of elements. Refer to Figure 2.6.

If we are processing a list α and want to process a sublist β of α, then we preserve the upper bound of α by encoding it in the relative order of elements starting from $A[right_\beta + 1]$. We can encode k similarly. We know how many bits to expect in the encoding of $right_\alpha$ and k, since $right_\alpha \leq c \cdot right_\beta$ and $k \leq right_\alpha$. Encoding $right_\alpha$ and k requires only $O(\log n)$ elements, so we clearly have enough elements to encode this information if n is sufficiently large. Furthermore, we can distinguish between the first and second subproblems by noting that $right_\alpha = c \cdot right_\beta$ if β corresponds to the first subproblem of α, and $right_\alpha \leq 4\, right_\beta$ if β corresponds to the second subproblem of α, and $c \geq 5$.

We can describe this optimized algorithm as follows. Let n_0 be some sufficiently large constant.

function OptDistinctISELECT(A,u,k)

1. (Initialization.)
   Set $u_0 \leftarrow u$.
2. (Handle special cases.)
   If $u < n_0$, then sort $A[1 \ldots u]$, set $x \leftarrow A[k]$, and go to 5.
3. (Solve the first subproblem.)
   Arrange $A[1 \ldots u]$ into $\lfloor \frac{u}{c} \rfloor$ scattered lists of size c, and place the median of each list in the front of the list.

4. Assume the list of medians occupies $A[1 \ldots u']$. Encode u in the relative order of $A[u'+1 \ldots u'+2\lceil \log cu' \rceil]$, and encode k in the relative order of $A[u'+1+2\lceil \log cu' \rceil \ldots u'+4\lceil \log cu' \rceil]$. Set $u \leftarrow u'$, $k \leftarrow \lceil \frac{u'}{2} \rceil$. Go to 2.

5. (The termination condition.)
   If $u = u_0$, then return x.

6. (The recovery step.)
   Set $u' \leftarrow u$ and recover u by from the relative order of $A[u'+1 \ldots u'+2\lceil \log cu' \rceil]$.

7. (Returned from second subproblem, so the current subproblem is finished.)
   If $u \leq 4u'$, then go to 5.

8. (Returned from first subproblem—solve the second subproblem.)
   Recover k from the relative order of $A[u'+1+2\lceil \log cu' \rceil \ldots u'+4\lceil \log cu' \rceil]$.

9. Partition $A[1 \ldots u]$ so that there is a position p such that $A[p] = x$, everything in $A[1 \ldots p-1]$ is less than x, and everything in $A[p+1 \ldots u-1]$ is greater than x.

10. If $k = p$, then go to 5.
    If $k < p$, then set $u' \leftarrow p-1$.
    If $k > p$, then reverse $A[1 \ldots u]$ and set $u' \leftarrow u-p$, $k \leftarrow k-p$.
    Encode u in the relative order of $A[u'+1 \ldots u'+2\lceil \log cu' \rceil]$. Set $u \leftarrow u'$. Go to 1.

end OptDistinctISELECT

The correctness of *OptDistinctISELECT* can be shown in the same manner as our previous correctness proof. It should be clear that *OptDistinctISELECT* performs a postorder traversal of the recursion tree of *RSELECT* and that the output x is propagated properly. It is not difficult to see that the rearrangement performed in steps 3, 4, 9, and 10 of our algorithm will make the subproblems solved by *OptDistinctISELECT* correspond exactly to the subproblems solved by *RSELECT*; that is, the elements associated with a subproblem of *OptDistinctISELECT* are the same as the elements associated with the corresponding subproblem of *RSELECT*. Furthermore, it is straightforward to show u and k are correctly

set on subproblem entry. Lastly, it is straightforward to show that u and k are correctly recovered, since all elements are distinct and the number of pairs used to encode u and k is a function of the size of the current subproblem.

We analyze the comparisons required by *OptDistinctISELECT* under the assumption that $c = 13$, which is the optimal value of c. Setting up the first subproblem now takes only $\frac{24}{13}n + 4\lceil \log n \rceil$ comparisons; recovering from the first subproblem takes only $4\lceil \log n \rceil$ comparisons; setting up the second subproblem requires only $n + 2\lceil \log n \rceil$ comparisons; and recovering from the second subproblem requires only $2\lceil \log n \rceil$ comparisons. Thus we have the recurrence relation

$$C(n) = C(\frac{n}{13}) + C(\frac{19}{26}n) + \frac{37}{13}n + 10\lceil \log n \rceil$$

We can straightforwardly show by induction that $C(n) \leq 14.8n + o(n)$. For the basis of the induction we sort the elements when $n < 64$; our encoding schemes for u and k are guaranteed to work when $n \geq 64$.

Therefore, only $14.8n + o(n)$ comparisons are required to solve the implicit selection problem with distinct elements. Notice that this is faster than the slow BFPRT algorithm! However, if step 1 of *RSELECT* is modified to perform selections rather than sorting, then *RSELECT* can be shown to require $14.8n$ comparisons. We can also show that the number of data movements required by *OptDistinctISELECT* is $O(n)$ as we did for *DistinctISELECT*.

Chapter 3

Implicit selection

3.1 New problems and their solutions

In this chapter we consider the implicit selection of the kth smallest of n arbitrary elements, that is, repetitions are allowed. We show how we can alter the algorithm of the previous chapter to solve this more general problem.

We saw in the last chapter that there are four potential sources of problems: avoiding recopying, saving endpoints, saving k, and implementing recursive calls. We can still avoid recopying using the technique presented in chapter 2. However, our technique for preserving k obviously fails since it relies on the relative order of pairs of elements. Our technique for preserving endpoints also fails, but for subtler reasons. Our ability to emulate recursive calls depends on our ability to recover endpoints, and so our technique for emulating recursive calls also fails.

The reason our endpoint encoding scheme fails is because we cannot properly nest lists using the encoding method described in the previous chapter. In Figure 3.1, note that we cannot recover $right_\beta$ while we are processing γ, because α_{max} happens to fall before position $right_\beta + 1$. The problem is that $\alpha_{max} \in \beta$, and our encoding scheme requires that for any list β, $\forall x \in \beta, A[left_\beta - 1] < x < A[right_\beta + 1]$.

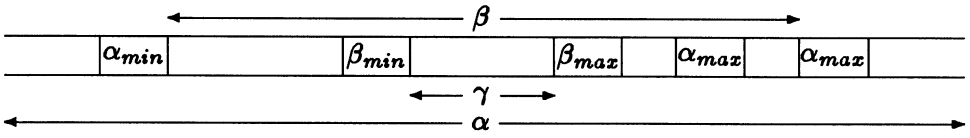


Figure 3.1: How the encoding scheme fails

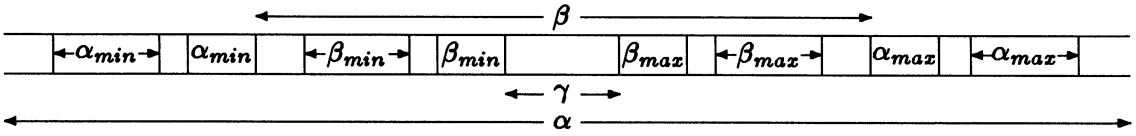


Figure 3.2: The corrected encoding scheme

The solution to this is straightforward. If we have a list α and a sublist β of α, then to ensure that $\alpha_{min} \notin \beta$ and $\alpha_{max} \notin \beta$, we find all elements equal to α_{min} and α_{max} and place them in contiguous blocks at the left and right ends of the list α; we then restrict β to being a sublist of the elements whose values are strictly between α_{min} and α_{max}. Figure 3.2 shows an example of the corrected encoding scheme. Theorem 2.1 implies the correctness of this scheme, since all invariants are satisfied except for element distinctness, and element distinctness is not necessary in the proof of the correctness of the endpoint recovery scheme. Thus our outward linear searches will suffice to find previous endpoints.

Before we proceed further, we need to introduce some new notation. We will denote the multiset of all elements equal to x by $x_=$. We will denote the ith smallest distinct element of a list α by α_i; that is, $\alpha_1 = \alpha_{min}$ and $\alpha_i = (\alpha - \bigcup_{j=1}^{i-1} \alpha_{=j})_{min}$ for $i > 1$. We will denote the ith largest distinct element of α by $\alpha_{max-i+1}$ for $i > 1$; that is, $\alpha_{max-i} = (\alpha - \alpha_{=max} - \bigcup_{j=1}^{i-1} \alpha_{=max-i})_{max}$.

To encode a rank k in a list α, we can no longer depend on the relative order of $2 \log \#\alpha$ elements. Note, however, that we can use a unary encoding scheme to encode k. Each value k corresponds to the position $left_\alpha + k - 1$, and that we can encode k by placing an

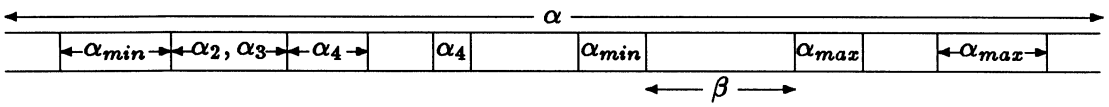


Figure 3.3: Encoding k

identifiable element in this position. To indicate k we can use α_2. Since α_2 may occur many times, we find all occurrences of α_2 and place them in a contiguous block beside the $\alpha_{=min}$ block. Let $\hat{\alpha}$ denote $\alpha - \alpha_{=min} - \alpha_{=2} - \alpha_{=max}$, that is, the sublist of α where all occurrences of α_{min}, α_2, and α_{max} have been removed. We encode k by swapping the rightmost α_2 with the element in position $left_\alpha + l - 1$. If this position is not in $\hat{\alpha}$, then the result of the selection is immediate.

We now need to select only from $\hat{\alpha}$. Unfortunately, during the solution of the first subproblem of α, our algorithm rearranges the middle $\frac{n}{c} + 2$ elements of $\hat{\alpha}$, and so α_2 cannot be stored there. However, we can map this region to the first $n/c + 2$ elements of $\hat{\alpha}$, and we can maintain an extra bit to determine whether α_2 has been relocated. To store this extra bit, we use the relative order of two blocks. We find all occurrences of α_3 and α_4 and place them beside α_{min} and α_2. $\hat{\alpha}$ will then be $\alpha - \alpha_{=min} - \alpha_{=2} - \alpha_{=3} - \alpha_{=4} - \alpha_{=max}$. We then use the relative order of blocks $\alpha_{=2}$ and $\alpha_{=3}$ to store the extra bit, and the rightmost element of $\alpha_{=4}$ to indicate k. See Figure 3.3.

To recover k, we skip over the leftmost block and determine the relative order of the next two blocks, which must be $\alpha_{=2}$ and $\alpha_{=3}$. Then we find the rightmost occurrence of the minimum of the remainder of the list; this must be α_4. From this position and the relative order of $\alpha_{=2}$ and $\alpha_{=3}$, we can recover k.

If we have returned from processing a list β to processing a list α, then we can determine whether β corresponds to the first or second subproblem of α by comparing $left_\beta$ and $right_\beta$ with $left_{\hat{\alpha}}$ and $right_{\hat{\alpha}}$. If β consists of the middle n/c elements of $\hat{\alpha}$, then we have returned

from the first subproblem. If $left_\beta = left_{\hat{\alpha}}$ or $right_\beta = right_{\hat{\alpha}}$, then we have returned from the second subproblem.

3.2 An implicit selection algorithm

3.2.1 The algorithm

Our algorithm can be described as follows. Let n_0 be some sufficiently large constant.

function ISELECT(A,l,u,k)

1. (Initialization.)
   Set $l_0 \leftarrow l$, $u_0 \leftarrow u$.
2. (Handle special cases.)
   If $u - l < n_0$, then sort $A[l \ldots u]$, set $x \leftarrow A[l + k - 1]$, and go to 9.
3. (Solve the first subproblem.)
   Let α denote $A[l \ldots u]$. Find and move all occurrences of α_{min}, α_2, α_3, and α_4 to contiguous blocks at the left end of α and all occurrences of α_{max} to the right end of α. If at any time we detect that there are fewer than 6 distinct elements, then set $x \leftarrow A[l + k - 1]$ and go to 9.
4. Let $\hat{l}$, $\hat{u}$ be the endpoints of the list $\hat{\alpha} = \alpha - \alpha_{=min} - \alpha_{=2} - \alpha_{=3} - \alpha_{=4} - \alpha_{=max}$. If $l + k - 1 < \hat{l}$ or $l + k - 1 > \hat{u}$, then set $x \leftarrow A[l + k - 1]$ and go to 9.
5. Arrange $A[\hat{l} \ldots \hat{u}]$ into $\lfloor \frac{\hat{u}-\hat{l}+1}{c} \rfloor$ scattered lists of size c, and place the median of each list in the middle of the list.
6. Let l', u' be the endpoints of the list of medians. Encode l and u by swapping $A[l] \leftrightarrow A[l' - 1]$, $A[u] \leftrightarrow A[u' + 1]$.

7. Encode k as follows. If $l + k - 1 < l' - 1$ or $l + k - 1 > u' + 1$, then swap $A[l + k - 1] \leftrightarrow A[\hat{l} - 1]$. Otherwise, invert the relative order of blocks $\alpha_{=2}$ and $\alpha_{=3}$, and swap $A[\hat{l} + (l + k - l')] \leftrightarrow A[\hat{l} - 1]$. (Note that $A[\hat{l} - 1]$ contains α_4.)

8. Set $l \leftarrow l'$, $u \leftarrow u'$, and $k \leftarrow \lceil \frac{u'-l'+1}{2} \rceil$. Go to 2.

9. (The termination condition.)
   If $l = l_0$ or $u = u_0$, then return x.

10. (The recovery step.)
    Set $l' \leftarrow l$, $u' \leftarrow u$. Recover l and u as follows. Find the index i of the first element to the left of $A[l' - 1]$ that is less than $A[l' - 1]$. If no such i exists, then set $l \leftarrow l_0$; otherwise, set $l \leftarrow i + 1$. Find the index j of the first element to the right of $A[u' + 1]$ that is greater than $A[u' + 1]$. If no such j exists, then set $u \leftarrow u_0$; otherwise, set $u \leftarrow j - 1$.

11. Swap $A[l' - 1] \leftrightarrow A[l]$, $A[u' + 1] \leftrightarrow A[u]$. Find $\hat{l}$ by skipping the first 3 blocks of $A[l \ldots u]$. Find $\hat{u}$ by skipping the block at the right end of $A[l \ldots u]$.

12. (Returned from second subproblem, so the current subproblem is finished.)
    If $l' = \hat{l}$ or $u' = \hat{u}$, then go to 9.

13. (Returned from first subproblem—solve the second subproblem.)
    Find the rightmost occurrence $A[m]$ and second rightmost occurrence $A[\tilde{l}]$, if any, of the minimum element of $A[\hat{l} \ldots \hat{u}]$. If $\min(A[\hat{l} \ldots \hat{u}])$ occurs only once, then set $\hat{l} \leftarrow \hat{l} + 1$; otherwise, set $\hat{l} \leftarrow \tilde{l} + 2$. If $\alpha_{=2}$ and $\alpha_{=3}$ are in ascending order, then set $k \leftarrow m - l + 1$; otherwise, set $k \leftarrow m - \hat{l} + l' - l$. Swap $A[m] \leftrightarrow A[\hat{l} - 1]$.

14. Partition $A[\hat{l} \ldots \hat{u}]$ so that there are positions p and q such that everything in $A[p \ldots q]$ is equal to x, everything in $A[\hat{l} \ldots p - 1]$ is less than x, and everything in $A[q + 1 \ldots \hat{u}]$ is greater than x.

15. If $p \leq l - k + 1 \leq q$, then go to 9.
    If $l - k + 1 < p$, then set $l' \leftarrow \hat{l}$, $u' \leftarrow p - 1$, $k \leftarrow k - (\hat{l} - l)$.
    If $l - k + 1 > q$, then set $l' \leftarrow q + 1$, $u' \leftarrow \hat{u}$, $k \leftarrow k - (q - 1 - l)$.

Encode l and u by swapping $A[l] \leftrightarrow A[l'-1]$, $A[u] \leftrightarrow A[u'+1]$. Set $l \leftarrow l'$, $u \leftarrow u'$. Go to 2.

end ISELECT

We show the correctness of *ISELECT* in a similar manner to the proof of correctness of our first algorithm. We show that *ISELECT* emulates a slightly modified version of *RSELECT*. More specifically, we consider a modified version *RSELECT3* of *RSELECT* in which we precede the first step by a step that finds and moves the blocks $\alpha_{=min}$, $\alpha_{=2}$, $\alpha_{=3}$, $\alpha_{=4}$, and $\alpha_{=max}$. It should be clear that *ISELECT* performs an postorder traversal of the recursion tree of *RSELECT3*, that rearrangement in *ISELECT* is effectively identical to that of *RSELECT3*, and that the output x is propagated correctly. Hence we only have to show that l, u, and k are properly set on entry to subproblems and are properly recovered.

Clearly steps 6 and 15 correctly set l, u, and k for subproblem entry. We thus only have to show that our recovery scheme is valid. Our encoding of l and u clearly satisfy conditions (i), (iii), and (iv) of Theorem 2.1. This implies that the searches performed in step 8 are sufficient to recover l and u.

k is indicated by the position of the rightmost occurrence of α_4 and the relative order of the blocks $\alpha_{=2}$ and $\alpha_{=3}$. The rightmost occurrence of α_4 and the relative order of $\alpha_{=2}$ and $\alpha_{=3}$ cannot be modified during the solution of the first subproblem, and k need not be encoded in the second problem. The encoding technique of step 7 and the recovery technique of step 13 can be shown to be consistent, so our recovery of k is correct.

3.2.2 Analysis

We count the number of comparisons required by the algorithm to select from a list α. Let n be $\#\alpha$, n_1 be the number of elements equal to α_{min}, n_i be the number of elements equal to α_i, for $i = 2, 3, 4$, and n_{max} be the number of elements equal to α_{max}.

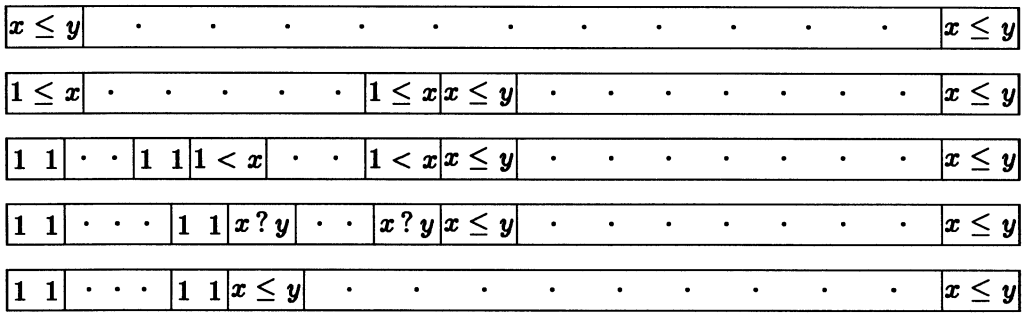


Figure 3.4: Finding all occurrences of 1

Clearly we can implicitly find and move all blocks $\alpha_{=1}, \ldots, \alpha_{=4}$ to the left end of α in $n + (n - n_1) + (n - n_1 - n_2) + (n - n_1 - n_2 - n_3) = 4n - 3n_1 - 2n_2 - n_3$ comparisons. However, we will describe a method by which $\alpha_{=1}, \ldots, \alpha_{=l}$ can be found and moved to the left end of α using only $\frac{l+1}{2}n + \sum_{i=1}^{l}(\frac{i-l+3}{2})n_l$ comparisons and $O(l)$ storage, for any l. This way we need only $\frac{5}{2}n + \frac{1}{2}n_2 + n_3 + \frac{3}{2}n_4$ comparisons to find and move blocks $\alpha_{=1}, \ldots, \alpha_{=4}$.

Our method works as follows. Suppose we have moved $\alpha_{=1}, \ldots, \alpha_{=i-1}$ to the left end of α, and the remainder of α consists of sorted pairs. Let $\overline{\alpha}$ be $\alpha - \alpha_{=1} - \cdots - \alpha_{=i-1}$. We can move all pairs of the form $(\alpha_i, x \ge \alpha_i)$ to the left end of $\overline{\alpha}$ using only $\frac{n-n_1-\cdots-n_{i-1}}{2}$ comparisons. We then move all pairs of the form (α_i, α_i) to the left end of the $\{(\alpha_i, x)\}$ block, which requires at most n_i comparisons. Now $\overline{\alpha}$ consists of three blocks $A = \{(\alpha_i, \alpha_i)\}$, $B = \{(\alpha_i, x > \alpha_i)\}$, and $C = \{(x > \alpha_i, y \ge x)\}$. We can move all elements of B greater than α_i to the right end of B without using any comparisons, so that $\overline{\alpha}$ consists of three blocks $A' = \alpha_{=i}$, $B' = \{(x, y?x)\}$, and C. We then sort each pair of B' so that $\overline{\alpha}$ consists of the blocks $\alpha_{=i}$ and $\{(x > \alpha_i, y \ge x)\}$; this requires at most $n_i/2$ comparisons. Then the left end of α will contain $\alpha_1, \ldots, \alpha_i$ and the remainder of α will consist of sorted pairs. Thus we can repeat this process for α_{i+1}. Figure 3.4 shows how the elements are rearranged if $\alpha_i = 1$.

Note that if there are fewer than $l + 1$ distinct elements, then we will find that $n_1 + \cdots + n_i = n$ for some $i \le l$.

We count the number of comparisons required by this algorithm as follows. Initially we must sort each pair of α, requiring $n/2$ comparisons. To find and move each block $\alpha_{=i}$ requires $\frac{n-n_1-\cdots-n_{i-1}}{2} + \frac{3}{2}n_i$ comparisons. Thus the total number of comparisons used to find and move all blocks $\alpha_{=1}, \ldots, \alpha_{=l}$ is

$$\frac{n}{2} + \sum_{i=1}^{l}[\frac{n}{2} - \sum_{j=1}^{i-1}\frac{n_j}{2} + \frac{3}{2}n_i] = (\frac{l+1}{2})n + \sum_{i=1}^{l}\frac{i-l+3}{2}n$$

This scheme can easily be extended to find the l smallest and m largest distinct elements using only $\frac{l+m+1}{2}n + \sum_{i=1}^{l+m}(\frac{i-l-m+3}{2})n$ comparisons.

When counting the number of comparisons required by *ISELECT*, we assume $c = 29$, which is the optimal value of c. To set up the first subproblem, we need $3n - \frac{1}{2}n_1 + 0n_2 + \frac{1}{2}n_3 + 1n_4 + \frac{3}{2}n_{max}$ comparisons to find and move $\alpha_{=min}$, $\alpha_{=2}$, $\alpha_{=3}$, $\alpha_{=4}$, and $\alpha_{=max}$; no comparisons to find $\hat{l}$ and $\hat{u}$; $\frac{70}{29}(n - n_1 - n_2 - n_3 - n_4 - n_{max})$ comparisons to find the medians of the lists of size 29; no comparisons to encode l and u; and no comparisons to encode k. Thus at most $\frac{157}{29}n - \frac{169}{58}n_1 - \frac{70}{29}n_2 - \frac{111}{58}n_3 - \frac{41}{29}n_4 - \frac{53}{58}n_{max}$ comparisons are required to set up the first subproblem.

To recover from the first subproblem, we require $n - \frac{n-n_1-n_2-n_3-n_4-n_{max}}{29}$ comparisons to find the preserved bounds; $n_1+n_2+n_3$ comparisons to locate $\hat{l}$ and determine the relative order of $\alpha_{=2}$ and $\alpha_{=3}$; n_{max} comparisons to locate $\hat{u}$; and $n-n_1-n_2-n_3-n_{max}$ comparisons to recover k. Thus $\frac{57}{29}n + \frac{n_1}{29} + \frac{n_2}{29} + \frac{n_3}{29} + \frac{n_4}{29} + \frac{n_{max}}{29}$ comparisons are required to recover from the first subproblem.

To set up the second subproblem, we need $n - n_1 - n_2 - n_3 - n_4 - n_{max}$ comparisons to partition $\hat{\alpha}$, and no comparisons to encode l and u. To recover from the second subproblem, we require $n - s_2(n)$ comparisons to recover previous endpoints, $n_1 + n_2 + n_3$ comparisons to find $\hat{l}$, and n_{max} comparisons to recover $\hat{u}$.

Clearly $C(n) > n$, so $C(n)$ is maximized by setting $s_2(n)$ to be $\frac{43}{58}(n - n_1 - n_2 - n_3 - n_4 - n_{max})$. Therefore the recurrence relation for the number of comparisons required by

ISELECT is

$$\begin{aligned} C(n) &= C(\frac{n-n_1-n_2-n_3-n_4-n_{max}}{29}) + T(\frac{43}{58}(n-n_1-n_2-n_3-n_4-n_{max})) \\ &\quad + \frac{501}{58}n - \frac{62}{29}n_1 - \frac{95}{58}n_2 - \frac{33}{29}n_3 - \frac{95}{58}n_4 - \frac{4}{29}n_{max} \end{aligned}$$

Clearly this is maximized when we set $n_1 = n_2 = n_3 = n_4 = n_{max} = 1$, so

$$C(n) \leq C(\frac{n-5}{29}) + C(\frac{43}{58}(n-5)) + \frac{501}{58}n$$

We can show by induction that $C(n) \leq \frac{501}{13}n < 38.5385n$. The basis for the induction is that we sort the elements for $n < 32$; our encoding schemes for l, u, and k are guaranteed to work for $n \geq 32$.

We can also show that the number of data movements required by *ISELECT* is $O(n)$.

3.3 Optimizations

3.3.1 The algorithm

As with the selection algorithm for distinct elements, we can optimize this algorithm by encoding only one endpoint. We rearrange elements such that the left endpoint of any list is always at the left end of the array, as in the previous chapter. However, we cannot encode u or k in the relative order of any elements, so we use an asymmetric version of the above scheme. We use α_{max}, α_{max-1}, α_{max-2}, and α_{max-3} to encode the upper endpoint and the rank k.

There is one complication, however. There is no way to distinguish between returning from the first subproblem and returning from the second problem by examining the size of the lists. Fortunately, the solution to this is simple. Currently we are using the relative order of α_{max-1} and α_{max-2} to determine whether we have displaced the rank indicator α_{max-3}. We can use the relative order of α_{max-1}, α_{max-2}, α_{max-3} to determine whether we

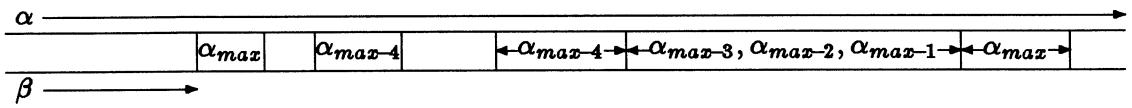


Figure 3.5: Using one endpoint

have displaced the rank indicator or whether we are returning from the second subproblem; we use α_{max-4} as the rank indicator.

If we want to return to a list α from a sublist β, then we can find the endpoint of α quickly if we place α_{max-1}, α_{max-2}, and α_{max-3} next to the endpoint of β. This way, we can quickly tell if we are returning from a first subproblem or a second subproblem. If we are returning from a first subproblem, then we know that $\#\alpha \geq c\#\beta$. If we are returning from a second subproblem, then we know that $\#\alpha \geq \frac{4}{3}\#\beta$. This information allows us to skip a large number of elements when searching for the endpoint of α.

Notice that we only use three of the six possible permutations of $\alpha_{=max-1}$, $\alpha_{=max-2}$, and $\alpha_{=max-3}$; either we are in the first subproblem and the rank indicator is not displaced, we are in the first subproblem and the rank indicator is displaced, or we are in the second subproblem. An optimization we can make is to restrict the rank indicator α_{max-4} to at most $n/5$ positions, and use five permutations of $\alpha_{=max-1}$, $\alpha_{=max-2}$, and $\alpha_{=max-3}$ to indicate the actual value of k. Figure 3.5 shows how the elements of α are rearranged by the optimized encoding scheme.

Our optimized algorithm works as follows. Let n_0 be some sufficiently large constant.

function OptISELECT(A,u,k)

1. (Initialization.)

   Set $u_0 \leftarrow u$.

2. (Handle special cases.)

If $u < n_0$, then sort $A[1 \ldots u]$, set $x \leftarrow A[k]$, and go to 10.

3. (Solve the first subproblem.)

   Let α denote $A[1 \ldots u]$. Find and move all occurrences of α_{max}, α_{max-1}, α_{max-2}, α_{max-3}, and α_{max-4} to contiguous blocks at the right end of α. If at any time we detect that there are fewer than 6 distinct elements, then set $x \leftarrow A[k]$ and go to 10.

4. Let $\hat{u}$ be the endpoint of the list $\hat{\alpha} = \alpha - \alpha_{=max} - \alpha_{=max-1} - \alpha_{=max-2} - \alpha_{=max-3} - \alpha_{=max-4}$. If $k > \hat{u}$, then set $x \leftarrow A[k]$ and go to 10.

5. Arrange $A[1 \ldots \hat{u}]$ into $\lfloor \hat{u}/c \rfloor$ scattered lists of size c, and place the median of each list in the front of the list.

6. Assume the list of medians occupies $A[1 \ldots u']$. Encode u by swapping $A[u] \leftrightarrow A[u' + 1]$.

7. Let $\overline{u}$ be the endpoint of the list $\alpha - \alpha_{=max}$. Rotate $A[u'+2 \ldots \overline{u}]$ so that $\alpha_{=max-3}$, $\alpha_{=max-2}$, and $\alpha_{=max-1}$ occupy $A[u'+1 \ldots u'']$, and $\alpha_{=max-4}$ occupies $A[u''' \ldots \overline{u}]$.

8. Encode k as follows. Let k' be k mod $\lceil \overline{u}/5 \rceil$. Swap $A[u'' + 1 + k'] \leftrightarrow A[u''']$. Change the relative order of blocks $\alpha_{=max-1}$, $\alpha_{=max-2}$, and $\alpha_{=max-3}$ depending on $\lfloor k/\lceil \frac{\overline{u}}{5} \rceil \rfloor$.

9. Set $u \leftarrow u'$, and $k \leftarrow \lceil \frac{u'}{2} \rceil$. Go to 2.

10. (The termination step.)

    If $u = u_0$, then return x.

11. (The recovery step.)

    Set $u' \leftarrow u$. Find the relative order of the three blocks in $A[u'+1 \ldots u'']$. Recover u by searching to the right of $A[(c-1)u' + u'']$ if we are returning from a first subproblem, or to the right of $A[\frac{u'}{2c/\lceil c/2 \rceil - 1} + u'']$ if we are returning from a second subproblem. If no such $u \leq u_0$ exists, then set $u \leftarrow u_0$.

12. Swap $A[u' + 1] \leftrightarrow A[u]$. If we have returned from the second subproblem, then go to step 10.

13. (Returned from first subproblem—solve the second subproblem.)

Let $\overline{u}$ be the endpoint of $\alpha - \alpha_{=max}$. Find the maximum element $A[m]$ of $A[u'' + 1 \ldots u'' + \frac{\overline{u}-u''}{5}]$. Recover k from $m - u''$ and the relative order of $\alpha_{=max-1}$, $\alpha_{=max-2}$, and $\alpha_{=max-3}$.

14. Search left from $\overline{u}$ to find the skip over the α_{max-4} block, if any. (Note $\alpha_{max-4} = A[m]$.) Swap $A[m]$ to the immediate left of the α_{max-4} block. Then rotate $A[u' + 2 \ldots \overline{u}]$ to place $\alpha_{=max-4}, \ldots, \alpha_{=max}$ in a contiguous block at right end of α. Let $\hat{u}$ be the endpoint of $\alpha - \alpha_{=max}\} - \cdots - \alpha_{=max-4}$.

15. Partition $A[1 \ldots \hat{u}]$ so that there are positions p and q such that everything in $A[p \ldots q]$ in equal to x, everything in $A[1 \ldots p]$ is less than x, and everything in $A[q+1 \ldots \hat{u}]$ is greater than x.

16. If $p \leq k \leq q$, then go to 10.
    If $k < p$, then set $u' \leftarrow p - 1$.
    If $k > q$, then reverse $A[1 \ldots \hat{u}]$ and set $u' \leftarrow u - q$, $k \leftarrow k - q$.

17. Encode u by swapping $A[u] \leftrightarrow A[u' + 1]$. Change the relative order of $\alpha_{=max-1}$, $\alpha_{=max-2}$, and $\alpha_{=max-3}$ to indicate that the second subproblem is being solved. Rotate $\alpha_{=max-1}$, $\alpha_{=max-2}$, and $\alpha_{=max-3}$ into the locations starting from $A[u'+2]$. Set $u \leftarrow u'$. Go to 2.

end OptISELECT

3.3.2 Correctness

To show correctness of *OptISELECT*, we prove a theorem showing that the one-endpoint recovery scheme is correct. Let l denote the number of lists.

Theorem 3.1 *Suppose we have an array A and indices* $u_1, \ldots, u_l$ *such that*

(i) $u_l < u_{l-1} < \cdots < u_1$ *(The lists are nested.)*

(ii) $\forall i, \forall j \leq u_i, A[j] < A[u_i + 1]$ *(To the immediate right of each list i is an element that is strictly greater than any element in list i.)*

(iii) $\forall i, (A[u_{i+1}+1] = max_{j \leq u_i}\{A[j]\})$ *(The element to the right of list $i+1$ is the maximum element of list i.)*

Then $\forall i < l, \forall j \ni u_{i+1} + 1 < j \leq u_i + 1, (A[j] > A[u_{i+1} + 1] \Longleftrightarrow j = u_i + 1)$.

This theorem implies that the recovery scheme of an asymmetric version of our endpoint encoding scheme is valid. The proof of this theorem is similar to the proof of Theorem 2.1.

Clearly *OptISELECT* performs an postorder traversal of a modified *RSELECT* that finds and moves $\alpha_{=max-4}, \ldots, \alpha_{=max}$. It should also be clear that the rearrangement performed ensures that the subproblems solved by *OptISELECT* correspond exactly to the subproblems solved in the modified *RSELECT*, that u and k are set correctly on subproblem entry, and that x is propagated correctly. We only have to show that u and k are correctly recovered.

The encoding of u satisfies the conditions of Theorem 3.1, so a linear search suffices to find u. We can skip elements when recovering u since the BFPRT algorithm guarantees that if we have returned from processing the first subproblem, then $u > cu'$, and if we have returned from the second subproblem, then $u > (\frac{2c}{2c-\lceil c/2 \rceil})u'$, where u' is the endpoint of the list we have returned from. Thus the recovery of u is correct.

k is encoded by placing α_{max-4} in one of $n/5$ possible locations and using five permutations of $\alpha_{=max-3}$, $\alpha_{=max-2}$, and $\alpha_{=max-1}$. The rank indicator α_{max-4} and the blocks $\alpha_{=max-3}, \ldots, \alpha_{=max-1}$ cannot be affected during the solution of the first subproblem, and k need not be encoded during the solution of the second subproblem. It can be shown that step 13 recovers k in a manner consistent with the encoding of k in step 8, so k is properly recovered.

3.3.3 Analysis

To count the number of comparisons used by *OptISELECT*, we assume $c = 29$; this is the optimal value of c. Let n_{max-4}, ..., n_{max-1}, n_{max} be the sizes of α_{max-4}, ..., α_{max-1}, α_{max}, respectively.

To set up the first subproblem, we need $3n + \frac{3}{2}n_{max-4} + 1n_{max-3} + \frac{1}{2}n_{max-2} + 0n_{max-1} - \frac{1}{2}n_{max}$ comparisons to find $\alpha_{max-4},\ldots,\alpha_{max}$; $\frac{70}{29}(n - n_{max} - n_{max-1} - n_{max-2} - n_{max-3} - n_{max-4})$ comparisons to find medians in step 4; none to encode u; and none to encode k.

To recover from the first subproblem, we need $n_{max-3} + n_{max-2} + n_{max-1}$ comparisons to find the relative order of three blocks; n_{max} comparisons to recover u; $(n - n_{max-3} - n_{max-2} - n_{max-1} - n_{max})/5$ comparisons to recover k; n_{max-4} comparisons to skip over α_{max-4}; and no comparisons to rotate a sublist of α.

To set up the second subproblem, we need n comparisons to partition the current list; and 3 comparisons to encode u.

To recover from the second subproblem, we need $n_{max-3} + n_{max-2} + n_{max-1}$ comparisons to find the relative order of three blocks; and $n - \frac{58}{43}s_2(n)$ comparisons to recover u.

Thus we have the recurrence relation

$$\begin{aligned} C(n) \;=\; & C(\tfrac{n - n_{max-1} - n_{max-2} - n_{max-3} - n_{max-4}}{29}) + C(s_2(n)) + 5n + \frac{70n}{29} + \frac{n}{5} \\ & + (\frac{5}{2} - \frac{70}{29})n_{max-4} + (\frac{14}{5} - \frac{70}{29})n_{max-3} + (\frac{23}{10} - \frac{70}{29})n_{max-2} \\ & + (\frac{9}{5} - \frac{70}{29})n_{max-1} + (\frac{3}{10} - \frac{70}{29})n_{max} - \frac{58}{43}s_2(n) + 3 \end{aligned}$$

where $s_2(n) \leq \frac{43}{58}(n - n_{max-1} - n_{max-2} - n_{max-3} - n_{max-4})$. Clearly $C(n)$ is maximized when $n_{max-3} = n_{max-2} = n_{max-1} = n_{max} = 1$. Since $C(s_2(n)) > s_n$, $C(n)$ is maximized when $s_2(n)$ is maximized. Also, since $C(\frac{n - \cdots - n_{max-4}}{29}) > \frac{3}{29}(n - \cdots - n_{max-4})$, we can show that $C(n)$ is maximized when $n_{max-4} = 1$. Thus we have the recurrence relation

$$C(n) \leq C(\frac{n}{29}) + C(\frac{43}{58}n) + 4n + \frac{70n}{29} + \frac{n}{5}$$

By induction, we can show that $C(n) \leq \frac{383.6n}{13} < 29.5077n$, if we sort when $n < 32$ as our basis; our encoding schemes for u and k are guaranteed to work for $n \geq 32$.

We can also prove that the number of data movements is $O(n)$.

Chapter 4

Improvements

4.1 Improvements

In the previous two chapters, we have presented some implicit selection algorithms based on the slow BFPRT algorithm. It was suggested [16] that one of the optimizations of the fast BFPRT algorithm be incorporated into our implicit selection algorithms. In particular, it was suggested that we sort lists of size c and maintain the invariant that lists of size c be sorted on entry to any subproblem. This seemingly minor change results in a considerable number of complications but dramatically reduces the number of comparisons required by the selection algorithms. In the remainder of this discussion, we will assume c is odd.

The improved algorithms which we will discuss are based on our optimized algorithms; they use only one endpoint. Our encoding schemes for u and k remain the same, but we rearrange elements in a completely different manner. In particular, we no longer use scattered lists. Instead, we use sorted, contiguous lists of size c. We can use sorted, scattered lists for the case when elements are distinct, but this can be shown to be of no advantage. Also, sorted, scattered lists do not readily allow us to obtain an efficient general case algorithm.

We will describe the improvements in detail for only the distinct element case. Our optimized algorithm consists of seven parts:

1. Initialization
2. First subproblem preparation—rearrangement
3. First subproblem preparation—encoding u and k
4. First subproblem recovery
5. Second subproblem preparation—rearrangement
6. Second subproblem preparation—encoding u and k
7. Second subproblem recovery

Our encoding and recovery schemes do not change at all, so we will only discuss the initialization and rearrangement steps.

4.1.1 Initialization

Because we require sorted, contiguous lists of size c on entry to any subproblem, we clearly must include a sorting step during initialization.

4.1.2 Rearrangement—first subproblem

In *OptDistinctISELECT*, we have to find medians and move them to the left end of the list. Our invariant ensures that we have sorted lists of size c, so median finding is trivial.

When moving the medians, note that the leftmost n/c elements will be disturbed. These n/c elements consist of n/c^2 medians and $\frac{n}{c} - \frac{n}{c^2}$ non-medians. The remaining elements consist of $\frac{n}{c} - \frac{n}{c^2}$ medians and $(\frac{c-1}{c})^2 n$ non-medians. To move the medians of all lists to

the leftmost n/c elements, we swap the non-medians of the disturbed elements with the medians of the remaining elements.

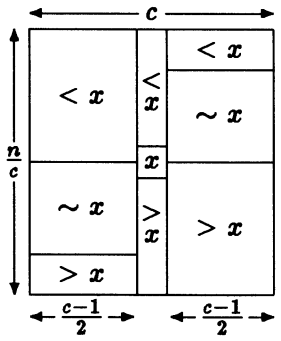
We also have to sort sublists of size c of the list of medians to maintain the invariant. Note that there is nothing analogous to this step in *OptDistinctISELECT*.

4.1.3 Rearrangement—second subproblem

When rearranging elements in the second subproblem, we must undo the swapping of the medians that we performed in the first subproblem. We swap the center elements of the rightmost $\frac{n}{c} - \frac{n}{c^2}$ lists of size c with the non-center elements of the leftmost n/c^2 lists of size c. It is unlikely that we will obtain the original list because the n/c medians will usually have been rearranged by the first subproblem. However, if the center of each list of size c were removed, then we know we have sorted lists of size $c - 1$.

Actually, this is not quite true. $O(\log n)$ elements may not be restored properly because of the binary encoding of u and k. More precisely, pairs of $O(\log n)$ elements are rearranged to encode u and k; when recovering u and k, we rearrange the $O(\log n)$ elements to obtain lists of size c such that each list contains two sorted lists of size $\frac{c-1}{2}$ and an element in the center of the list. The centers of these lists may no longer contain the original contents of some of the leftmost n/c elements that were displaced during the first subproblem. Nevertheless, we can swap the center elements of these lists anyway; $O(\log n)$ of the leftmost n/c^2 elements may then be out of order, but we can sort these $O(\log n)$ elements to solve this problem.

We now have n/c lists of size c; each list of size c consists of two sorted lists of size $\frac{c-1}{2}$, and an element in the center that is the median of some list. We want to find the rank of x and discard elements. To find the rank of x, we first note that x is the median of the list of medians. We then perform a binary search on the non-medians, that is, we perform a binary search on each sorted list of size $\frac{c-1}{2}$. Note that we do not have to do this for all the lists. All the elements of at least half of the left lists of size $\frac{c-1}{2}$ are less than x, and



$> x$ indicates that all the elements of the list are greater than x
$\sim x$ indicates that the list has elements less than and greater than x
$< x$ indicates that all the elements of the list are less than x

Figure 4.1: The improved rearrangement procedure

all the elements of at least half of the right lists of size $\frac{c-1}{2}$ are greater than x, left and right being with respect to Figure 4.1. Since we are dealing with sorted lists, these cases can be recognized in one comparison. Also, we can distinguish a left list from a right list by analyzing the endpoints of the list. Thus for at least half the lists, we need to perform only one comparison.

We now want to discard some elements. We relax the discard condition; instead of discarding all the elements greater than x or all the elements less than x, we discard the medians greater than x and the lists of size $\frac{c-1}{2}$ whose elements are all greater than x, or we discard the medians less than x and the lists of size $\frac{c-1}{2}$ whose elements are all less than x, respectively. We are guaranteed to discard at least $\frac{\lceil c/2 \rceil n}{2c}$ elements, so this does not worsen the worst case.

The problem with discarding elements is that we need to know the rank of x before we know whether to discard elements greater than x or less than x. To discard the medians greater than x or the medians less than x, we perform a modified quicksort partition of the scattered list of size n/c of medians. To discard lists of size $\frac{c-1}{2}$, we rearrange the lists while scanning them, such that the lists whose elements are all less than x occupy the upper

positions, the lists whose elements fall on both sides of x occupy the middle positions, and the lists whose elements are all greater than x occupy the lower positions, upper and lower being with respect to Figure 4.1.

We now form lists of size c by merging left and right lists of size $\frac{c-1}{2}$ and then performing binary insertions of the retained medians. If the number of retained left and right lists of size $\frac{c-1}{2}$ differ by more than one, then we even out the number of left and right lists by swapping retained left lists with discarded right lists or vice versa. If there are retained medians left over after merging lists, we can form additional sorted, contiguous lists of size c from the medians. If we exhaust the retained medians when forming lists of size c, then we can steal elements from a retained list of size $\frac{c-1}{2}$.

After forming the lists of size c, all of the retained elements will be in one large contiguous block in the left or right end of the current list, left and right being with respect to the one-dimensional representation of the list. If the retained elements fall in the right end, then we rotate the elements so that the retained elements fall in the left end of the array.

All of these steps ensure that we discard at least $\frac{\lceil c/2 \rceil}{2c}n$ elements, that the retained elements are in the left end of the current list, and that the retained elements consist of sorted, contiguous lists of size c.

4.2 Analysis

We count the number of comparisons required by the above algorithm as follows. We use the Ford-Johnson algorithm [9] for sorting. We assume $c = 43$, which is the optimum value of c.

The cost of the algorithm is

$$C(n) = C_0(n) + C'(n)$$

where $C_0(n)$ is the cost of initialization and $C'(n)$ is the cost of the main part of the algorithm. The cost of initialization is simply the cost of sorting lists of size 43; 177

comparisons are required by the Ford-Johnson algorithm to sort 43 elements, so $C_0(n) = \frac{177}{43}n$. We can easily derive a recurrence relation for $C'(n)$:

$$C'(n) = C'(\frac{n}{c}) + C'((1 - \frac{\lceil c/2 \rceil}{2c})n) + f_{1,p}(n) + f_{1,r}(n) + f_{2,p}(n) + f_{2,r}(n)$$

where $f_{i,p}(n)$ is the cost of setting up the ith subproblem and $f_{i,r}(n)$ is the cost of recovering from the ith subproblem, for $i = 1, 2$.

To set up the first subproblem, we have to find the medians of the sorted lists, requiring no comparisons; sort lists of size c of the list of medians, requiring $\frac{177}{43} \cdot \frac{n}{43}$ comparisons; and encode u and k, requiring $O(\log n)$ comparisons. Thus $\frac{177n}{43^2} + O(\log n)$ comparisons are required to set up the first subproblem.

To recover from the first subproblem, we have to recover u and k, requiring $O(\log n)$ comparisons; and correct the ordering of some elements, requiring $O(\log n)$ comparisons. Thus $O(\log n)$ comparisons are required.

To set up the second subproblem, we have to search the $2\frac{n}{c}$ lists of size $\frac{c-1}{2}$; for at least n/c lists, we need only 1 comparison each; for the other lists, we need at most $2 + \lfloor \log(\frac{c-1}{2}) \rfloor$ for the failed comparison and the binary search. We also have to partition the list of medians, requiring n/c comparisons. We then have to build lists of size c. Discarded elements are never considered when building the lists, and building a list of size c requires merging two lists of size $\frac{c-1}{2}$ and performing a binary insertion into a list of size $c - 1$. Thus the cost of building the lists of size c is at most $(1 - \frac{\lceil c/2 \rceil}{2c})n \cdot \frac{1}{c}[c - 1 + \lfloor \log(c - 1) \rfloor]$. The total cost of setting up the second subproblem is thus $\frac{8}{43}n + \frac{32}{43} \cdot \frac{47}{43}n$.

The cost of recovering from the second subproblem is $O(\log n)$.

Thus,

$$\begin{aligned} C'(n) &= C'(\frac{n}{c}) + C'((1 - \frac{\lceil c/2 \rceil}{2c})n) + \frac{177n}{43^2} + \frac{8n}{43} + \frac{32 \cdot 47n}{43^2} + O(\log n) \\ &= C'(\frac{n}{43}) + C'(\frac{32n}{43}) + \frac{177n}{43^2} + \frac{8n}{43} + \frac{32 \cdot 47n}{43^2} + O(\log n) \end{aligned}$$

By induction we can show that $C'(n) \leq \frac{405}{86}n + o(n)$. Thus

$$C(n) = C_0(n) + C'(n) \leq \frac{177}{43}n + \frac{405}{86}n + o(n) = \frac{759}{86}n + o(n) < 8.8256n + o(n)$$

We can also count the number of data movements required by our algorithm; the worst case number of data movements is $O(n)$.

4.3 The general case algorithm

We can optimize the general case algorithm discussed in chapter 3 in exactly the same way as described above. This time, however, we do not have the problem of the encoding of u and k interfering with the relocation of the list of medians. Another change from the optimized algorithm of chapter 3 is that we now use more than three blocks to distinguish the first subproblem from the second subproblem.

The analysis of the general case algorithm is similar to the distinct element selection algorithm. We choose $c = 85$, which can be shown to be the optimal value of c. However, we need to find all occurrences of the l largest distinct elements, for some constant l. Rather than using sorted pairs as we did in chapter 3, we can easily modify our block selection algorithm to take advantage of the sorted lists of size c. In fact, we need only $\frac{l \cdot n}{c} + c_{max-l+1}n_{max-l+1} + \cdots + c_{max}n_{max}$ comparisons to find all occurrences of the l largest distinct elements. Another change from the distinct element case is that we have no $O(\log n)$ terms. Instead, we require $\frac{n}{(l-2)!-1}$ comparisons to recover k after solving the first subproblem.

Note that $c_{max-l+1}, \ldots, c_{max}$ are not constants and depend on the positions of all occurrences of the l largest distinct elements. To simplify the analysis, we want to ensure that the comparisons required by the algorithm is maximized when $n_{max-l+1} = \cdots = n_{max} = 1$. Unfortunately, $c_{max-l+1}, \ldots, c_{max}$ can be quite large because each occurrence of one of the l largest distinct elements may force a binary insertion to be performed on each

list of size $c - 1$. However, if we require that each element has at most a constant number of occurrences, then $c_{max-l+1}n_{max-l+1} + \cdots + c_{max}n_{max}$ is at most a constant. If we let $l = 7$, we can show that the number of comparisons required by our improved algorithm is $\frac{229029}{24395}n + o(n) < 9.38836n + o(n)$.

For completely arbitrary elements, we let $c = 21$, and we use four blocks to encode k, that is, we let $l = 6$. This way, for each occurrence of one of the six largest distinct elements, at worst we require 1 comparison to search for the element and 5 comparisons to perform a binary insertion of the element. We can show that we save more than 6 comparisons per element because of the reduced size of the subproblems, so the worst case occurs when the number of occurrences of the six largest distinct elements is minimized. We can then show that the number of required comparisons is $\frac{250}{23}n < 10.8696n$.

We can also count the number of data movements required. We can prove that $O(n)$ data movements are used by the general case algorithm in the worst case.

Chapter 5

An application—updating implicit k-d trees

5.1 Representation

A k-d tree, or multidimensional binary search tree, is a binary tree where each node is a vector of k keys and is associated with an integer between 0 and $k-1$ inclusive; this integer is called the *discriminator* of the node. We denote the keys of node x by $K_0(x), \ldots, K_{k-1}(x)$. The invariant maintained in a k-d tree is that for any node x with a discriminator j, if y is in the left subtree of x then $K_j(y) < K_j(x)$, and if y is in the right subtree of x then $K_j(y) > K_j(x)$. To handle equality of the keys, Bentley suggested that a superkey $S_j(x) = K_j(x)K_{j+1}(x)\cdots K_{k-1}(x)K_0(x)\cdots K_{j-1}(x)$ be used consisting of the cyclical concatenation of all keys starting with K_j rather than K_j alone.

We will choose the discriminator of every node at level l to be l mod k. We also define a node y to be j-greater than a node x if $S_j(y) > S_j(x)$, and to be j-less than x if $S_j(y) < S_j(x)$. It is clear how these definitions can be used to define j-maximum, j-minimum, and j-median.

Bentley discussed an algorithm for constructing an optimized, balanced k-d tree from a set of n elements in $O(n \log n)$ time. His algorithm can be described as follows.

function Optimize(set A, discriminator j):pointer p

1. If A is null, return nil.
2. Set $p \leftarrow j$-median of A.
3. Set A_L to be the subset of elements of A that are j-less than p, and A_R to be the subset of elements of A that are j-greater than p.
4. Set Left(p) $\leftarrow$ Optimize(A_L, (j+1) mod k), and Right(p) $\leftarrow$ Optimize(A_R, (j+1) mod k). Return p.

end Optimize

An implicit data structure is a data structure that is stored in the first n locations of a semi-infinite one-dimensional array, where n is the number of elements in the data structure such that only $O(1)$ pointers need be maintained. Integer indices in the range $[0, n]$ are considered to be equivalent to pointers.

Munro [17] devised a static, implicit representation of k-d trees based on Bentley's construction algorithm. He ensures that each subtree t occupies a contiguous set of locations $l \ldots r$. The root of t occupies the middle position $\lceil \frac{l+r}{2} \rceil$, the left subtree of t occupies the left half $l \ldots \lceil \frac{l+r}{2} \rceil - 1$, and the right subtree of t occupies the right half $\lceil \frac{l+r}{2} \rceil + 1 \ldots r$.

A left-complete tree is a binary tree of height h in which each leaf is at level $h-2$ or $h-1$, and if a node in the tree has a right descendant at level $h-1$, then all its left descendants which are leaves are also at level $h-1$. We consider the level of the root to be 0 and the height of a tree of 1 node to be 1.

Our data structure is essentially an implicit representation of left-complete k-d trees. We store such a tree in breadth-first order as in a heap [26], so that the children of a node

at location i occupy locations $2i$ and $2i+1$. Left-complete k-d trees can be constructed in a manner similar to Bentley's *Optimize* algorithm. Because the shape of a left-complete tree of n nodes is fixed, we can easily determine the number of nodes n_L in the left subtree of the root. Rather than selecting the j-median as the root, we select the (n_L+1)th smallest element with respect to the j-discriminator. We do not need to use recursion; we can traverse the binary tree in breadth-first order and rebuild at each node. Our construction algorithm works as follows.

procedure Construct(array A, number of elements n)

$j \leftarrow 0$;
$l \leftarrow 1$;
$r \leftarrow 1$;
while $l \leq n$ do
    for i from l to $\min(r, n)$ do
        let n_L be the number of nodes in the left subtree of $A[i]$;
        find m such that $A[m]$ is the (n_L+1)th j-smallest element of the subtree
            rooted at $A[i]$;
        swap $A[i] \leftrightarrow A[m]$;
        move the elements that are j-less than $A[i]$ into the left subtree of $A[i]$, and
            the elements that are j-greater than $A[i]$ into the right subtree of $A[i]$;
    end for;
    $l \leftarrow 2l$;
    $r \leftarrow 2r+1$;
    $j \leftarrow (j+1) \bmod k$;
end while;

end Construct

This construction takes at most $O(n \log n)$ time. Each iteration of the while loop corresponds to a level, so there are at most $\log n$ iterations of the while loop. The for loop iterates through each node x in current level, and each iteration basically performs a selection and a redistribution. Both of these operations can be done in time proportional to the number of nodes in the subtree rooted at x. Thus in each iteration of the while loop, the work done by the for loop is proportional to the number of nodes in the subtrees rooted at any node in the current level. The nodes in the subtrees rooted at any node in the current level are simply the nodes at or below the current level, so clearly each iteration of the while loop requires $O(n)$ time.

In a practical implementation of this construction algorithm, an implicit, $O(n)$ expected time selection algorithm would be more suitable that our implicit, $O(n)$ worst case selection algorithm.

5.2 Queries

k-d trees can support various types of queries efficiently. Bentley gave an algorithm capable of answering any intersection query. An intersection query specifies a region and returns all elements of the k-d tree that lie in the specified region.

The simplest type of intersection query is an *exact match query*, that is, given a vector of k keys, is it in the k-d tree? Exact match queries can be answered in $O(\log n)$ time with balanced k-d trees, as follows.

function ExactMatch(array A, number of elements n, record p)

$i \leftarrow 1$; (i is the current position.)
$j \leftarrow 0$; (j is the current discriminator.)
while $i \leq n$ do
    if $A[i] = p$ then return true;

```
        if p is j-less than A[i]
            then set i ← 2i
            else set i ← 2i + 1;
        set j ← (j + 1) mod k;
    end while;
    return false;
end ExactMatch
```

In a *partial match query*, we specify a value for t of k keys, where $t < k$. If $s_1, \ldots, s_t$ are the keys specified and $v_1, \ldots, v_t$ are the key values specified, then the result of the query is $\{p : K_{s_i}(p) = v_{s_i} \text{ for } 1 \leq i \leq t\}$. Bentley showed that partial match queries can be solved in $O(tn^{1-t/k})$ time in the worst case.

In a *range query*, we specify an interval $[l_i, u_i]$ for each key K_i. The set of elements that we are searching for is $\{p : l_i \leq K_i(p) \leq u_i \text{ for } 1 \leq i \leq k\}$. Lee and Wong [14] showed that range queries can be answered in $O(kn^{1-1/k})$ time in the worst case. Silva-Filho [23] showed that range queries still require $O(n^{1-1/k})$ time in the average case for balanced k-d trees.

In a *partial range query*, we specify a range of values for t of k keys, where $t < k$. If $s_1, \ldots, s_t$ are the keys specified and $[l_1, u_1], \ldots, [l_t, u_t]$ are the key intervals specified, then the result of the query is $\{p : l_{s_i} \leq K_{s_i}(p) \leq u_{s_i} \text{ for } 1 \leq i \leq k\}$. Lee and Wong showed that partial range queries can be solved in $O(tn^{1-1/k})$ time in the worst case.

Bentley gave an algorithm for solving any type of intersection query including those defined above. His algorithm does not specifically need to know the type of region that it is searching; instead, it is passed two functions that describe the region. The first function *InRegion* is passed a node and returns true if and only if that node is in the desired region. The second function *BoundsIntersectRegion* is passed a bounds array and returns true if and only if the hyperrectangle described by the bounds intersects the desired region. His

routine also uses a procedure *Found* to report all nodes found in the region. To facilitate general region searches, Bentley suggested the use of a bounds array B. The bounds array B associated with a node p has $2k$ entries such that for any descendant q of p, $B[2j] \leq K_j(q) \leq B[2j+1]$.

His recursive algorithm works as follows:

procedure RegionSearch(node P, bounds array B)

1. If InRegion(P) then call Found(P).
2. Set $B_L \leftarrow B$ and $B_H \leftarrow B$. Let j be the discriminator of node P. Set $B_L[2j+1] \leftarrow K_j(P)$, $B_R[2j] \leftarrow K_j(P)$.
3. If Left(P)$\neq$nil and BoundsIntersectRegion(B_L), then RegionSearch(Left(P),B_L).
4. If Right(P)$\neq$nil and BoundsIntersectRegion(B_R), then RegionSearch(Right(P),B_H).

end RegionSearch

For our structure we can make an implicit emulation of this algorithm that does not require additional storage for temporary copies of B. To avoid using temporary copies of B, we encode previous values of B in the k-d tree.

procedure ImplicitRegionSearch(array A)

1. $i \leftarrow 1$.
2. If InRegion($A[i]$) then call Found(P).
3. (Move down in the tree.)
   Let j be the discriminator of node $A[i]$.

   If $2i \leq n$ then (try to go left)

swap $B[2j+1] \leftrightarrow K_j(A[i])$;
if BoundsIntersectRegion(B) then
set $i \leftarrow 2i$; go to 2
else
swap $B[2j+1] \leftrightarrow K_j(A[i])$;
end;
elsif $2i+1 \leq n$ then (try to go right)
swap $B[2j] \leftrightarrow K_j(A[i])$;
if BoundsIntersectRegion(B) then
set $i \leftarrow 2i+1$; go to 2
else
swap $B[2j] \leftrightarrow K_j(A[i])$;
end;
end.

4. (Move up in the tree.)
Set $i' \leftarrow i$, $i \leftarrow \lceil \frac{i}{2} \rceil$. If $i = 0$, then return. Otherwise, let j be the discriminator of node $A[i]$.

If $i = 2i'$ then
swap $B[2j+1] \leftrightarrow K_j(A[i])$;
if $2i+1 \leq n$ then
swap $B[2j] \leftrightarrow K_j(A[i])$;
if BoundsIntersectRegion(B) then
set $i \leftarrow 2i+1$; go to 2
else
swap $B[2j] \leftrightarrow K_j(A[i])$;
end;
end;

```
        go to 4;
    else
        swap B[2j] ↔ K_j(A[i]);
        go to 4;
    end.
end ImplicitRegionSearch
```

We claim that this algorithm is a nonrecursive emulation of the recursive *RegionSearch* algorithm. First, we claim that the nonrecursive algorithm performs a preorder traversal of the tree. Step 2 processes the current node. If the current node is not a leaf, then step 3 moves downward, always choosing the left child over the right child if the left child exists. Step 4 moves up in the tree, and continues moving up if the routine has moved up from a right child or the routine has moved to a node that has no right child; otherwise the routine moves to the right and goes downward again. The routine terminates when it tries to move upward from the root. Clearly this corresponds to a preorder traversal.

Clearly B and i are correctly set when moving downward, and i is correctly restored when moving upward, so we only have to show that B and A are restored correctly when we move upward through the tree. Suppose we are at node $A[i]$ with discriminator j. If we want to search the left subtree of $A[i]$, then we encode the current value of B by swapping $B[2j+1] \leftrightarrow K_j(A[i])$. If we want to search the right subtree, then we swap $B[2j] \leftrightarrow K_j(A[i])$. In either case we will always change $K_j(A[i])$. When we travel up to node i with discriminator j, we know that we must swap $K_j(A[i])$ with an element in B. If we moved up from a left child, then we know that $B[2j+1]$ was swapped; if we moved up from a right child, then $B[2j]$ was swapped. Thus when we travel up to a node, we will always know how to correctly restore that node and the bounds array B associated with that node.

5.3 Updates

To handle updates, we essentially use a degenerate form of a partial rebuilding scheme suggested by Overmars [20]; our scheme is degenerate in that we have to rebuild part of the tree after almost every update.

To insert x in a left-complete k-d tree, we perform an exact match query for x. If x is already present, then the update is redundant. Otherwise, consider the final value of the index where the search failed. We call this the natural insertion position, because x satisfies all the invariants of the k-d tree if it is inserted here. We place x in position $n+1$ of the array; we call this the required insertion position. We then find the lowest common ancestor a of the natural and required insertion positions and reconstruct the subtree rooted at a. This is equivalent to inserting x at its natural insertion position and relocating it to its required insertion position, and then rebuilding the subtree rooted at a.

Deletions are handled similarly. To delete x from a left-complete k-d tree, we again perform an exact match query. If x is not present, the deletion is redundant. Otherwise, we replace x with the element in position n; the natural deletion position is n, and the required deletion position is the position of x. We take the lowest common ancestor a of the natural and required deletion positions and reconstruct the subtree rooted at a. Thus we relocate the node at position n from its natural deletion position to its required deletion position and rebuild the subtree rooted at a.

In general, when updating we displace a node from its natural position, where it satisfies the partial order of the k-d tree, to its required position. We then rebuild the subtree rooted at the lowest common ancestor of the natural and required update positions.

Clearly our method of handling updates preserves left-completeness. We ensure that position $n+1$ is filled in the case of insertions and position n is cleared in the case of deletions. Furthermore, the left and right subtrees of any left-complete tree is also left-complete, and our reconstruction algorithm cannot change the shape of any left-complete

tree.

We only need to show that our update algorithms satisfy the invariants of the k-d tree. Suppose x is the natural position of the displaced node, y is the required position of the node, and LCA(x, y) is the lowest common ancestor of x and y. We claim LCA(x, y) is the highest node where the invariants fail, so that rebuilding the subtree rooted at LCA(x, y) will satisfy the invariants of the k-d tree. To see this, note that if z is an ancestor of LCA(x, y) and $z \neq$ LCA(x, y), then the x and y must fall in the same subtree of z, and so the invariants of the k-d tree do not fail at z. Rebuilding at any descendant of LCA(x, y) is not sufficient to satisfy the invariants of the k-d tree since the invariants fail at LCA(x, y).

The worst-case update time is clearly $\Theta(n \log n)$. Our reconstruction routine requires $\Theta(n \log n)$ time to rebuild n nodes, and at worst we have to rebuild the entire tree. However, at any time we can make an insertion require $\Omega(n \log n)$ time by ensuring that the required insertion position and natural insertion position fall on opposite subtrees of the root. This will make the root the lowest common ancestor, and thus we must reconstruct the entire tree. We can trivially make deletions require $\Omega(n \log n)$ time by deleting the root. Thus for any sequence of operations, we can make all operations require $\Theta(n \log n)$ time, and the amortized worst-case time is $\Theta(n \log n)$.

Worse yet, the expected update time is also $O(n \log n)$, since for a random insertion we have a 1 in 2 chance that the required and natural insertion positions fall on opposite subtrees of the root, and for a random deletion there is also a 1 in 2 chance that the required and natural deletion positions fall on opposite subtrees of the root. Nevertheless, our approach is superior to reconstructing the entire tree after every update. We can show that if the cost of rebuilding n nodes is at most $cn \log n$, then the expected insertion time and the expected deletion time is about $\frac{2}{3}cn \log n$.

For convenience, we define *leftmost*(i) to be the leftmost node at level i. Consider a complete tree of height h with $2^{h+1} - 1$ nodes. There are 2^{h+1} possible natural insertion positions since there are 2^{h+1} external nodes, and the required insertion position is the

leftmost external node at level $h+1$. Clearly the ancestors of the required insertion position are just *leftmost*(i), for $i = 0, \ldots, h$. The number of external nodes in the right subtree of *leftmost*(i) is 2^{h-i}, so the probability that the lowest common ancestor of the required and natural insertion positions being *leftmost*(i) is 2^{-1-i}. The number of nodes in the subtree rooted at *leftmost*(i) is 2^{h+1-i}, including the inserted node, so the expected insertion time is

$$\sum_{i=0}^{h} 2^{-1-i}[c \cdot 2^{h+1-i} \log(2^{h+1-i})] = c \sum_{i=0}^{h} 2^{h-2i} \log(2^{h+1-i}) \leq c \cdot 2^h \log(2^{h+1}) \sum_{i=0}^{h} 2^{-2i}$$
$$\leq c \cdot 2^h \log(2^{h+1}) \sum_{i=0}^{\infty} 2^{-2i} \leq c \cdot 2^h \log(2^{h+1})(\frac{1}{1-\frac{1}{4}}) = \frac{2}{3} c \cdot 2^{h+1} \log(2^{h+1}) = \frac{2}{3} cn \log n$$

For the case of deletion, consider a left-complete tree of height $h+1$ with 2^{h+1} nodes. The ancestors of the required deletion position are again *leftmost*(i), for $i = 0, \ldots, h$. The number of nodes in the right subtree of *leftmost*(i) is $2^{h-i} - 1$, so there are 2^{h-i} natural deletion positions that result in *leftmost*(i) being the lowest common ancestor of the natural and required deletion positions: either the leftmost node itself may be deleted, or any node in its right subtree may be deleted. The probability of *leftmost*(i) being the lowest common ancestor is thus 2^{-1-i}. The number of nodes in the subtree rooted at *leftmost*(i) is $2^{h+1-i}-1$, so by a similar analysis to the case of insertion, we find that the expected cost of deletion is $\frac{2}{3} cn \log n$.

We can, however, improve the amortized worst-case cost of insertion without worsening the expected deletion time by batching insertions together. For instance, we can batch $\log n$ insertions together before rebuilding the k-d tree, so that the amortized insertion time will be $O(n)$. More sophisticated dynamization schemes are possible.

In conclusion, our implementation of k-d trees is extremely storage efficient; our representation requires only a constant amount of extra space because it is implicit, and our update and query algorithms require at most a constant amount of additional space. Also, our implementation can support many types of queries efficiently, and we have presented

an algorithm for general intersection queries. Finally, the expected update costs of our data structure are superior to that of continual reconstruction of the entire tree.

Chapter 6

Conclusions

6.1 Conclusions and Open Problems

We have shown that the implicit selection problem can be solved in $8.8256n + o(n)$ comparisons if all elements are distinct, $9.3884n + o(n)$ comparisons if every element can appear at most a constant number of times, and $10.8696n$ comparisons for arbitrary elements.

This naturally leads to the question of whether the implicit selection problem can be solved in less than $9n$ comparisons in general. There are various ways to improve the general case algorithm [15]. One method avoids using blocks to distinguish between the first and second subproblems by taking advantage of the fact that these blocks are used to encode only a constant number of bits. Since the maximum depth of the recursion is $O(\log n)$, this information can be stored in a constant number of pointers and does not have to be encoded in the array on which we are performing a selection. Another method for improving the general case algorithm is to use a binary encoding scheme for k. This has the complication of finding unequal pairs of elements, but we can avoid finding and moving blocks used to encode k and distinguish between the first and second subproblems.

Another method of improving the upper bounds is to emulate faster $O(n)$ worst case selection algorithms, such as the fast BFPRT algorithm or the Schönhage-Paterson-Pippenger

(SPP) algorithm. The fast BFPRT algorithm requires only $5.4305\dot{n}$ comparisons, but does not appear to be well-suited to being emulated implicitly; however, there may perhaps be an efficient method of doing so. It is not clear at all how the SPP algorithm can be efficiently emulated implicitly.

Other open problems lie in the area of lower bounds. Clearly all of the lower bounds on the selection problem apply to the implicit selection problem. Sharper lower bounds may be obtainable.

Bibliography

[1] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8:244–251, 1979.

[2] J. L. Bentley. Decomposable searching problems I: Static to dynamic transformations. *Journal of Algorithms*, 1:301–358, 1980.

[3] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, SE-5:333–340, 1979.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[5] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[6] Lewis Carroll. Lawn tennis tournaments. *St. James's Gazette*, 5–6, August 1, 1883.

[7] B. Durian. Quicksort without a stack. In *Mathematical Foundations of Computer Science*, pages 283–289, Springer-Verlag, 1986.

[8] R. W. Floyd and R. L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18:165–172, 1975.

[9] L. R. Ford and S. M. Johnson. A tournament problem. *American Mathematical Monthly*, 66:387–389, 1959.

[10] A. Hadian and M. Sobel. *Selecting the t-th largest using binary errorless comparisons.* Technical Report 121, Univ. of Minnesota, 1969.

[11] C. A. R. Hoare. Algorithm 63, Partition. *Communications of the ACM*, 4:321, 1961.

[12] C. A. R. Hoare. Algorithm 65, Find. *Communications of the ACM*, 4:321–322, 1961.

[13] S. S. Kislitsyn. On selection of the kth element of an ordered set by pairwise comparisons. *Sibirskii Mat. Zhurnal*, 5:557–564, 1964.

[14] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9:23–29, 1977.

[15] A. Lubiw. Private communication.

[16] J. I. Munro. Private communication.

[17] J. I. Munro. A multikey search problem. In *Proc. Allerton*, pages 241–244, 1979.

[18] J. I. Munro. Searching a two key table under a single key. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 383–387, 1987.

[19] J. I. Munro and H. Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21:236–250, 1980.

[20] M. H. Overmars. *The Design of Dynamic Data Structures.* Volume 156 of *Lecture Notes in Computer Science*, Springer-Verlag, 1983.

[21] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13:184–199, 1976.

[22] J. Schreier. On tournament elimination systems. *Mathesis Polska*, 7:154–160, 1932.

[23] Y. V. Silva-Filho. Average case analysis of region search in balanced k-d trees. *Information Processing Letters*, 8:219–223, 1979.

[24] J. van Leeuwen and D. Wood. Dynamization of decomposable searching problems. *Information Processing Letters*, 10:51–56, 1980.

[25] J. van Leeuwen and D. Wood. *Interval heaps.* Technical Report, Univ. of Waterloo, 1988. To appear.

[26] J. W. J. Williams. Algorithm 232, Heapsort. *Communications of the ACM*, 7:347–348, 1964.